

# **Problems in Cloud Security, Access Control and Logic Locking**

by

**Nahid Juma**

A thesis  
presented to the University of Waterloo  
in fulfillment of the  
thesis requirement for the degree of  
Doctor of Philosophy  
in  
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

©Nahid Juma 2020

# Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Philip W. L. Fong Associate Professor, CS, University of Calgary
Supervisor	Mahesh Tripunitara Professor, ECE, University of Waterloo
Internal Member	Stephen Smith Associate Professor, ECE, University of Waterloo
Internal Member	Arie Gurfinkel Associate Professor, ECE, University of Waterloo
Internal-External Member	Ian Goldberg Professor, CS, University of Waterloo

## **Author's Declaration**

This thesis consists of material all of which I authored or co-authored; see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

# Statement of Contributions

The work on cloud scheduling in Chapter 3 is published in IEEE Transactions of Dependable and Secure Computing [1]. IEEE requires that I include the following statement in my thesis [2]:

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of University of Waterloo's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to [http://www.ieee.org/publications\\_standards/publications/rights/rights\\_link.html](http://www.ieee.org/publications_standards/publications/rights/rights_link.html) to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

IEEE's policy on reusing published articles for a thesis is as follows [2]:

You may reuse your published article in your thesis or dissertation without requesting permission, provided that you fulfill the following requirements depending on which aspects of the article you wish to reuse.

Text excerpts: Provide the full citation of the original published article followed by the IEEE copyright line: ©20XX IEEE. If you are reusing a substantial portion of your article and you are not the senior author, obtain the senior author's approval before reusing the text.

Graphics and tables: The IEEE copyright line (©20XX IEEE) should appear with each reprinted graphic and table.

The work on forensic analysis for access control in Chapter 4 is undergoing review at the ACM Conference on Computer and Communications Security [3].

The work on logic locking in Chapter 5 is undergoing review at ACM Transactions on Privacy and Security [4].

I am first author of [1] and [3], and second author of [4]. My exact contributions are explicitly mentioned in a footnote at the beginning of each of Chapters 3, 4 and 5.

# Abstract

In this thesis, we study problems related to security in three different contexts: cloud scheduling, access control, and logic locking to protect digital ICs.

The first set of problems relates to security in cloud computing. Prior work suggests that scheduling, with security as a consideration, can be effective in minimizing information leakage, via side-channels, that can exist when virtual machines (VMs) co-reside in clouds. We analyze the overhead that is incurred by such an approach. We first pose and answer a fundamental question: is the problem tractable? We show that the seemingly simpler sub-cases of initial placement and migration across only two equal-capacity servers are both intractable (**NP**-hard). However, a decision version of the general problem to which the optimization version is related polynomially is in **NP**. With these results as the basis, we make several other contributions. We revisit recent work that proposes a greedy algorithm for this problem, called Nomad. We establish that if  $\mathbf{P} \neq \mathbf{NP}$ , then there exist infinitely many classes of input, each with an infinite number of inputs, for which a decrease in information leakage is possible, but Nomad provides none, let alone minimize it. We establish also that a mapping to Integer Linear Programming (ILP) in prior work is deficient in that the mapping can be inefficient (exponential-time), and therefore does not accurately convey the overhead of such an approach that actually decreases information leakage. We present our efficient reductions to ILP and boolean satisfiability in conjunctive normal form (CNF-SAT). We have implemented these approaches and conducted an empirical assessment using the same ILP solver as prior work, and a SAT solver. Our analytical and empirical results more accurately convey the overhead that is incurred by an approach that actually provides security (decrease in information leakage).

The second set of problems relates to access control. We pose and study forensic analysis in the context of access control systems. Forensics seeks to answer questions about past states of a system, and thereby provides important clues and evidence in the event of a security incident. Access control deals with who may perform what action on a resource and is an important security function. We argue that access control is an important context in which to consider forensic analysis, and observe that it is a natural complement of safety analysis,

which has been considered extensively in the literature. We pose the forensic analysis problem for access control systems abstractly, and instantiate it for three schemes from the literature: a well-known access matrix scheme, a role-based scheme, and a discretionary scheme. In particular, we ask what the computational complexity of forensic analysis is, and compare it to the computational complexity of safety analysis for each of these schemes. We observe that in the worst-case, forensic analysis lies in the same complexity class as safety analysis. We consider also the notion of logs, i.e., data that can be collected over time to aid forensic analysis. We present results for sufficient and minimal logs that render forensic analysis for the three schemes efficient. This motivates discussions on goal-directed logging, with the explicit intent of aiding forensic analysis. We carry out a case-study in the realistic setting of a serverless cloud application, and observe that goal-directed logging can be highly effective. Our work makes contributions at the foundations of information security, and its practical implications.

The third set of problems relates to logic locking to protect digital integrated circuits (ICs) against untrusted semiconductor foundries. We make two sets of complementary contributions, all rooted in foundations and bolstered by implementations and empirical results. Our first set of contributions regards observations about prior schemes and attacks, and our second is a new security notion. Towards the former, we make two contributions. (a) We revisit a prior approach called XOR-locking that has been demonstrated to be susceptible, in practice, to a particular attack called the SAT attack. We establish that (i) there exist circuits that are invulnerable to the SAT attack when XOR-locked with even a 1-bit key, and, (ii) there is a particular property that is inherent to benchmark circuits that explains why the SAT attack is successful against XOR-locked versions of those. Both (i) and (ii) are rooted in computing foundations: for (i), one-way functions; for (ii), average-case computational complexity, specifically, the class **distP**. (b) We revisit a state-of-art logic locking approach called TTLock whose generalization called SFLL-HD has been argued to be “provably secure” in prior work. We devise a new, probabilistic attack against TTLock. We explain, from foundations, why benchmark circuits that are locked using TTLock are susceptible to our new attack. Our observations (a) and (b), and prior work on attacks, informs our second contribution, which is a new security notion. Our notion is at least as strong as the property that underlies the SAT attack.

# Acknowledgements

I would like to express my deepest gratitude and appreciation to my advisor, Professor Mahesh Tripunitara, for guiding me, and for helping me develop my research skills over the past few years.

Thank you to Mohamed El Massad for sharing his expertise on logic locking, to Jonathan Shahan who I enjoyed working with on the cloud scheduling project, and to Professors Philip Fong, Ian Goldberg, Arie Gurfinkel and Stephen Smith, for graciously serving on my PhD Committee.

As this thesis was written during the global pandemic, COVID-19, I would like to acknowledge and thank the front-line workers who compromise their well-being so that the rest of us can maintain a sense of normality in our lives.

Finally, I will always be thankful to my parents and to my husband, Shaneabbas, for their unwavering love and support.



*To my wonderful sons, Mahdi & Abbas.*

# Table of Contents

<b>List of Figures</b>	<b>xiv</b>
<b>List of Tables</b>	<b>xvi</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background on Computational Complexity</b>	<b>4</b>
2.1 Definitions . . . . .	4
2.2 Computational Complexity Classes . . . . .	8
2.3 Average-Case Complexity . . . . .	9
<b>3 Overhead from Combating Side-channels in Cloud Systems by VM-Scheduling</b>	<b>11</b>
3.1 Introduction . . . . .	11
3.2 Minimizing Information Leakage . . . . .	17
3.3 Computational Complexity . . . . .	23
3.3.1 An Upper-Bound: <b>NP</b> . . . . .	23

3.3.2	A Lower-Bound: <b>NP</b> -hard . . . . .	24
3.3.3	Closed Migration under the $\langle R, C \rangle$ Model . . . . .	31
3.3.4	Mitigating NP-hardness and Open Problems . . . . .	33
3.4	Analysis of Prior Approaches . . . . .	36
3.4.1	Mapping to ILP . . . . .	36
3.4.2	Initial Design: Baseline Greedy . . . . .	37
3.4.3	Final Design: Nomad . . . . .	38
3.5	Reductions to ILP and CNF-SAT . . . . .	47
3.5.1	Reduction to ILP . . . . .	49
3.5.2	Reduction to CNF-SAT . . . . .	50
3.5.3	Empirical Evaluation . . . . .	53
3.6	Related Work . . . . .	56
3.7	Conclusions . . . . .	59
<b>4</b>	<b>Forensic Analysis for Access Control</b>	<b>60</b>
4.1	Introduction . . . . .	60
4.2	Problem . . . . .	67
4.2.1	Access Control Schemes and Systems . . . . .	67
4.2.2	Forensic Analysis: Problem Formulation . . . . .	69
4.2.3	Trusted Users . . . . .	72

4.3	Computational Complexity . . . . .	73
4.3.1	HRU . . . . .	74
4.3.2	ARBAC . . . . .	78
4.3.3	Graham-Denning . . . . .	85
4.4	Goal-directed Logging . . . . .	91
4.5	Case Study . . . . .	96
4.5.1	Background . . . . .	97
4.5.2	Forensic Analysis . . . . .	101
4.6	Related Work . . . . .	105
4.7	Conclusions . . . . .	107
<b>5</b>	<b>Logic Locking: Observations from Foundations and a Security Notion</b>	<b>108</b>
5.1	Introduction . . . . .	108
5.2	XOR-Locking . . . . .	113
5.2.1	The Effectiveness of XOR-Locking . . . . .	114
5.2.2	The Ineffectiveness of XOR-Locking . . . . .	118
5.3	TTLock . . . . .	125
5.3.1	Observations on TTTLock’s Security . . . . .	126
5.3.2	A New Attack on TTTLock . . . . .	128
5.4	A Security Notion . . . . .	133

5.5	Related Work . . . . .	138
5.6	Conclusions . . . . .	141
<b>6</b>	<b>Conclusions</b>	<b>142</b>
6.1	On VM-Scheduling to Combat Cloud Side-channels . . . . .	142
6.2	On Forensic Analysis for Access Control . . . . .	143
6.3	On Logic Locking to Protect Digital ICs . . . . .	144
	<b>Bibliography</b>	<b>146</b>
	<b>Appendices</b>	<b>159</b>
A	Forensic Analysis for Access Control . . . . .	160
A	An Upper-bound on Hardness for General ARBAC . . . . .	160
B	Graham-Denning with Trusted Users . . . . .	164

# List of Figures

3.1	Scheduling actions that can mitigate information leakage. . . . .	13
3.2	Client-to-client leakage under four information leakage models. . . . .	20
3.3	Example showing lack of optimal substructure in scheduling problem. . . . .	35
3.4	Example for which Nomad provides no security. . . . .	40
3.5	Example for which Nomad provides no security in the $\langle R, C \rangle$ model. . . . .	47
3.6	Plot showing empirical result from running example in Figure 3.5. . . . .	54
3.7	Plot showing Nomad, ILP and CNF-SAT's performances on 10 random initial placements. . . . .	54
3.8	Plot showing scalability of Nomad, ILP and CNF-SAT. . . . .	55
4.1	Forensic analysis vs. safety analysis. . . . .	61
4.2	Example showing how a state-change rule can be 'inverted'. . . . .	63
4.3	An example of an access control system in the HRU scheme [65]. . . . .	65
4.4	An example of an ARBAC system. . . . .	78

4.5	Set of commands that constitute the state-change rule for Graham-Denning based systems [70]. . . . .	87
4.6	An example of an access control system in the Graham-Denning scheme [80].	88
4.7	An example AWS policy. . . . .	97
4.8	An example CloudTrail log. . . . .	99
4.9	Portion of Hello, Retail!’s workflow. . . . .	100
5.1	Overview of logic locking. . . . .	109
5.2	Example of how a circuit is XOR-Locked. . . . .	113
5.3	Example of circuit for which XOR-Locking is effective. . . . .	114
5.4	Example illustrating the TTLock (or SFLL-HD <sup>0</sup> ) scheme [123]. . . . .	125
5.5	Plot of number of DIPs for SAT attack against TTLocked benchmarks [119].	126

# List of Tables

3.1	The symbols used for calculating total information leakage. . . . .	19
3.2	The inputs to the scheduling problem. . . . .	22
4.1	A summary of how the size of logs changes when goal-directed logging is used.	104
5.1	Time it takes for our wire redundancy checking procedure to finish executing on the set of benchmark circuits from prior work [115]. . . . .	124
5.2	Number of trials out of 100 where our attack recovers the designer's function for circuits from the MCNC and ISCAS'89 benchmarks. . . . .	129



# Chapter 1

## Introduction

In this thesis we use computing foundations to study computational problems related to information security in three different contexts.

Our first line of contribution is in the context of cloud computing. In particular, we consider the vulnerability that arises when a cloud provider hosts the virtual machines (VMs) of multiple clients on the same physical machine using virtualization. Although this multi-tenant setting results in economies of scale for the provider, it gives rise to a security threat because cross-VM side-channel attacks may occur — it is possible for a malicious client who is co-resident with a victim client to extract information about the victim by exploiting the underlying shared hardware. Prior work proposes models to quantify information leakage from cross-VM side-channels and suggests that a possible defence is to use a placement algorithm for virtual machines that minimizes information leakage under these models [5]. We study this optimization problem in more depth.

Our second line of contribution is in the context of access control systems. In access control systems, principals request for accesses to resources, and their requests are granted or denied based on the current authorization state. The authorization state may change over time, with changes possibly being effected by semi-trusted or untrusted users. We pose and study a new problem in this context — that of asking queries about the system’s past authorization states.

Answers to these forensic questions may expose or explain vulnerabilities in the access control system.

Our third line of contribution is in the context of logic locking to protect digital integrated circuits (ICs). Logic locking is a process used by a semiconductor designer to hide their intellectual property in a circuit before sending the circuit to an untrusted foundry for fabrication. While there have been several proposals for how to lock a circuit, to the best of our knowledge, each of these has been shown to be vulnerable to some attack or the other (see Section 5.5 in Chapter 5 for a discussion). We seek to gain further insight to comprehend this discouraging state of affairs in the logic locking research space.

Although each of our three lines of contributions in this work lie in different contexts, our approach to addressing them is common — we appeal to computing foundations. In particular, we use computational complexity theory to gain meaningful insights about the problems. Computational complexity theory is a subfield of theoretical computer science; its primary goal is to classify and compare the practical difficulty of solving problems about finite combinatorial objects. It is concerned with how much resources are required to solve a given computational task [6].

Our thesis statement is the following:

There exist problems in information security for which an analysis based in computational complexity can provide meaningful insights into trade-offs and thereby guide us to appropriate solution approaches.

We prove the thesis statement by construction. That is, we study three problems in information security and we demonstrate how computational complexity theory provides insights into those problems that were not previously known.

This thesis is organized as follows. In Chapter 2 we introduce definitions and preliminary material related to computational complexity. We study the cloud VM-scheduling problem in Chapter 3, the forensic analysis problem for access control in Chapter 4, and logic locking to protect digital ICs in Chapter 5. Each of Chapters 3, 4 and 5 provides background information

and a summary of prior work related to the corresponding security problem. Chapter 6 contains our conclusions.

## Chapter 2

# Background on Computational Complexity

In this chapter we provide definitions and explain the concepts related to computational complexity theory that are used in this thesis.

### 2.1 Definitions

A *computational problem* is a function that we seek to compute using a computational device such as a Turing machine. Thus, such a problem can be characterized by specifying: (i) inputs, and, (ii) outputs that correspond to those inputs. A particular specification of an input to a computational problem is called an *instance* of the problem. For example, Prime Factorization corresponds to the function whose domain is the set of positive integers  $\geq 2$ , and codomain is the powerset of prime numbers. The function maps an input integer to its set of prime factors. An example of an instance of Prime Factorization is 9, and the output of the function for this instance is  $\{3, 3\}$ .

A decision problem is one which corresponds to a function whose codomain is  $\{\text{false}, \text{true}\}$ ,

i.e., a bit. An example of a decision version of the Prime Factorization problem is one in which the input is a pair of positive integers  $\langle N, k \rangle$ , such that  $N, k \geq 2$ , and the output is ‘true’ if  $N$  has a prime factor  $\leq k$ , and ‘false’ otherwise.

*Solving a problem* means computing the function that corresponds to the problem. An *algorithm* is a step-by-step procedure for solving a problem. An algorithm is said to solve a problem if it can be applied to any instance of the problem and is guaranteed to produce the correct output for that instance. A *deterministic algorithm* is an algorithm which given a particular instance as input, always produces the same output. That is, its behaviour on an input instance can be completely predicted from the instance. In contrast, a *non-deterministic algorithm* is an algorithm with more than one allowed step at certain times and which always takes the right or best step. Conceptually, a non-deterministic algorithm could run on a deterministic computer with an unlimited number of parallel processors. Each time more than one step is possible, new processes are instantly forked to try all of them. When one process successfully finishes, its result is returned. Thus the computation is as fast as if it always chooses the right step. In this thesis, when we use the term ‘algorithm’ without qualification, we mean a deterministic algorithm.

The description of a problem instance that we provide as input to an algorithm can be viewed as a finite string of symbols chosen from a finite input alphabet. Although there are many ways to map instances to strings, for simplicity and without any loss of generality, we can assume that each problem has a fixed encoding scheme which maps its instances into strings describing them. The *input length* of an instance of a problem is used as a formal measure of the size of the instance, and is defined to be the number of symbols,  $n$ , in the description of the instance obtained from the problem’s encoding scheme.

There could be multiple algorithms which solve a problem but in general, we are interested in the most *efficient* one. In its broadest sense, the notion of efficiency, refers to all the various computing resources needed to execute an algorithm such as time and space. However, time requirements are often the most dominant factor, and by the ‘most efficient algorithm’ one normally means the fastest one. The *time complexity function* of an algorithm expresses its time requirements by giving, for each possible input length  $n$ , the largest amount of time

needed by the algorithm to solve a problem instance of that size.

The notations  $O(\cdot)$ ,  $\Omega(\cdot)$ ,  $\Theta(\cdot)$ ,  $o(\cdot)$  and  $\omega(\cdot)$  are used to compare functions. We use them to analyze the asymptotic efficiency of an algorithm, that is, how a particular resource required by an algorithm, such as its running time, increases with the input size  $n$  in the limit, as  $n$  increases without bound. We deal only with functions of the form  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ , that is only those functions whose domain is positive integers, and co-domain is positive reals. The reason is that an input size for us is always a positive integer, and time- and space-efficiency is quantified as a positive real.

For a given function  $g(n)$  we denote by  $O(g(n))$  the set of functions

$$O(g(n)) = \{f(n) : \exists \text{ positive constants } n_0 \text{ and } c \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

For a function  $f(n) \in O(g(n))$  it is customary to write  $f(n) = O(g(n))$ . The mindset, when we say  $f(n) = O(g(n))$  is that  $g(n)$  is an asymptotic upper-bound for  $f(n)$ .

For a given function  $g(n)$  we denote by  $\Omega(g(n))$  the set of functions

$$\Omega(g(n)) = \{f(n) : \exists \text{ positive constants } n_0 \text{ and } c \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}.$$

When we say  $f(n) = \Omega(g(n))$ , we mean that  $f(n) \in \Omega(g(n))$  and that  $g(n)$  is an asymptotic lower-bound for  $f(n)$ .

For a given function  $g(n)$ ,  $\Theta(g(n))$  is the set of functions that are in both  $O(g(n))$  and  $\Omega(g(n))$ . When we say  $f(n) = \Theta(g(n))$ , we mean that  $f(n) \in \Theta(g(n))$  and that  $g(n)$  is an asymptotic tight bound for  $f(n)$ .

The upper-bound provided by  $O(\cdot)$  notation may or may not be asymptotically tight. We use  $o(\cdot)$  to denote an upper-bound that is not asymptotically tight. Formally for a given function  $g(n)$  we denote by  $o(g(n))$  the set of functions

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0 \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}.$$

Similarly, the lower-bound provided by  $\Omega(\cdot)$  notation may or may not be asymptotically tight. We use  $\omega(\cdot)$  to denote a lower-bound that is not asymptotically tight. Formally for a given function  $g(n)$  we denote by  $\omega(g(n))$  the set of functions

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0 \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}.$$

We say that an algorithm is polynomial-time if there exists a constant  $k$  such that the algorithm's time-efficiency can be meaningfully characterized as  $O(n^k)$ , where  $n$  is the size of the input. We say that an algorithm is exponential-time if there exists a constant  $k > 1$  such that the algorithm's time-efficiency can be meaningfully characterized as  $O(k^n)$ , where  $n$  is the size of the input.

The *computational complexity* of a problem refers to the inherent difficulty of the problem. It characterizes the resources required to solve the problem. Problems are classified based on their inherent difficulty with certain relationships existing between classes.

We use the notion of a *reduction* to compare two problems from the standpoint of their computational complexity.

A *Cook reduction*, also called a *polynomial-time Turing reduction*, is a particular kind of reduction from problem  $A$  to problem  $B$ . We say problem  $A$  Cook-reduces to problem  $B$  if there exists an algorithm that solves problem  $A$  using a polynomial number of calls to an algorithm that solves problem  $B$ , and polynomial time outside of those calls.

A *Karp reduction*, also called a *polynomial-time many-one reduction*, is another kind of reduction. We say that a decision problem  $A$  Karp-reduces to a decision problem  $B$  if there

exists a polynomial-time computable function  $f$  that transforms instances of  $A$  into instances of  $B$  such that  $x$  is a ‘yes’ instance of  $A$  if and only if  $f(x)$  is a ‘yes’ instance of  $B$ .

In this thesis, unless otherwise specified, the term ‘reduction’ refers to a Karp reduction.

## 2.2 Computational Complexity Classes

We now define the computational complexity classes that are of interest to us.

The class of *decidable* problems comprises the decision problems for each of which there exists an algorithm that terminates and is correct for every instance of the problem.

The class of *undecidable* problems comprises the decision problems for each of which it is proven that there cannot exist an algorithm that terminates and is correct for every instance of the problem.

The class **P** is the set of decision problems for each of which there exists a constant  $k$  and an algorithm whose time-efficiency is  $O(n^k)$ . That is, **P** comprises decision problems that can be solved by an algorithm in polynomial time in the input size.

The class **NP** is the set of decision problems for each of which there exists a constant  $k$  and a non-deterministic algorithm whose time-efficiency is  $O(n^k)$ . An equivalent definition is that a decision problem is in the class **NP** if for its ‘yes’ instances there exists an efficiently sized proof that can be verified in polynomial time.

The class **PSPACE** is the set of decision problems for each of which there exists a constant  $k$  and an algorithm whose space-efficiency is  $O(n^k)$ .

The class **NPSpace** is the set of decision problems for each of which there exists a constant  $k$  and a non-deterministic algorithm whose space-efficiency is  $O(n^k)$ .

The class **EXPTIME** is the set of decision problems for each of which there exists a constant  $k$  and an algorithm whose time-efficiency is  $O(2^{n^k})$ .



The relationship between the above classes is:

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME} \subseteq \text{decidable}.$$

It is known that  $\mathbf{PSPACE} = \mathbf{NPSPACE}$  [7], and also that  $\mathbf{P} \subset \mathbf{EXPTIME}$ . Whether  $\mathbf{P} = \mathbf{NP}$  is a major open problem, although it is generally conjectured that  $\mathbf{P} \neq \mathbf{NP}$ . Whether  $\mathbf{NP} = \mathbf{PSPACE}$  and whether  $\mathbf{PSPACE} = \mathbf{EXPTIME}$  are also open problems.

When we say that a decision problem belongs to a complexity class, that is an expression of an upper-bound for the hardness of the problem. For example, if a decision problem is in  $\mathbf{NP}$ , that expresses that the problem is no harder than  $\mathbf{NP}$ . However it doesn't say anything about the lower-bound hardness of the problem.

The notion of a reduction can be used to express the lower-bound hardness of a problem relative to a complexity class. We say that a problem  $A$  is  $\mathcal{C}$ -hard under a type of reduction, where  $\mathcal{C}$  is a complexity class, if every problem in  $\mathcal{C}$  can be reduced to  $A$  using that type of reduction. For example, we may say that a problem  $A$  is  $\mathbf{NP}$ -hard under a Cook reduction. When one says a problem  $A$  is  $\mathcal{C}$ -hard without specifying under what kind of reduction, one usually means under a Karp reduction.

The notion of *completeness* combines an upper- and lower-bound on hardness. We say a problem  $A$  is  $\mathcal{C}$ -complete for a complexity class  $\mathcal{C}$  under a type of reduction if two conditions hold: (1)  $A$  is in  $\mathcal{C}$ , and (2)  $A$  is  $\mathcal{C}$ -hard under that type of reduction.

## 2.3 Average-Case Complexity

The computational complexity classes defined in the previous section relate to worst-case complexity. That is, a problem belongs to a particular class if its worst-case instance requires the resources characterized by that class. For example, a problem is in the class  $\mathbf{P}$  if its worst-case input can be solved in polynomial time.

As previously mentioned, a customary assumption is that  $\mathbf{P} \neq \mathbf{NP}$ . Therefore if a problem

is **NP**-hard, it is unlikely that an efficient algorithm exists that solves every instance of it. Because if such an algorithm exists, it would imply that all problems in **NP** can be solved in polynomial time and that **NP** = **P**. For many **NP**-hard problems however, the instances that arise in practice do not elicit the worst-case. Hence algorithm designers may try to design efficient algorithms that work for ‘many’ or ‘most’ instances. Similarly for problems in certain fields like cryptography, it is often not sufficient to demonstrate that the problem faced by the attacker is hard in the worst-case. Rather, what is needed is evidence that the average instance that arises in practice is hard for the attacker. This motivates the theory of average-case complexity.

Average-case complexity classes characterize the difficulty of distributional problems. A *distributional problem* is a pair  $\langle \mathcal{L}, \mathcal{D} \rangle$ , where  $\mathcal{L}$  is a decision problem and  $\mathcal{D}$  is a distribution on instances of  $\mathcal{L}$ .

The class **distP** is the average-case analogue of the class **P**. Informally, the class **distP** comprises the distributional problems that can be solved by an algorithm that is efficient on average over the particular distribution. Formally the class **distP** is defined as follows [6]:

Let  $\mathcal{D}$  be a distribution over  $\{0, 1\}^*$ ; we denote sampling from a distribution using a left arrow,  $\leftarrow$ . If  $A$  is an algorithm, let  $\text{time}_A(x)$  denote the running-time of algorithm  $A$  on input  $x$ . Let  $E[\cdot]$  denote expectation, and  $|s|$  denote the size of the encoding of a string  $s$ .

Given a decision problem  $\mathcal{L}$  and a distribution  $\mathcal{D}$  we say that  $\langle \mathcal{L}, \mathcal{D} \rangle \in \mathbf{distP}$  if there exists an algorithm  $A$  for  $\mathcal{L}$  and positive constants  $c, \epsilon$  such that:

$$E_{x \leftarrow \mathcal{D}} \left[ \frac{\text{time}_A(x)^\epsilon}{|x|} \right] \leq c$$

The above formalization was chosen over naive formalizations, such as the one that requires the expectation of  $\text{time}_A(x)$  to be bounded by a constant, because it does not suffer from issues that are suffered by the naive formalizations. These issues include not being closed under the functional composition of algorithms, being dependent on the computational model adopted, and being dependent on the encoding used [6].

## Chapter 3

# The Overhead from Combating Side-channels in Cloud Systems by VM-Scheduling<sup>1</sup>

### 3.1 Introduction

Cloud computing has become an important paradigm by offering flexibility and cost-effectiveness to both providers of infrastructure and services, and their clients. An aspect of cloud computing is virtualization, which enables providers to host virtual machines (VMs) of multiple clients on the same physical infrastructure. Indeed, multiple VMs may be co-resident, that is, reside on the same server. While co-residency has benefits, such as economies of scale, it gives rise to a serious security threat. Information may leak from a

---

<sup>1</sup>Substantial portions of this chapter are from previously published work [1]. My contributions are the computational complexity analysis for the scheduling problem (Section 3.3), the analysis of the three approaches taken by prior work [5] (Section 3.4), and the reductions of the scheduling problem to ILP and CNF-SAT (Sections 3.5.1 and 3.5.2). The empirical evaluation of methods based on these two reductions (Section 3.5.3) was conducted by Jonathan Shahan and further details about the evaluation can be found in our paper [1] and in Jonathan's PhD dissertation [8].

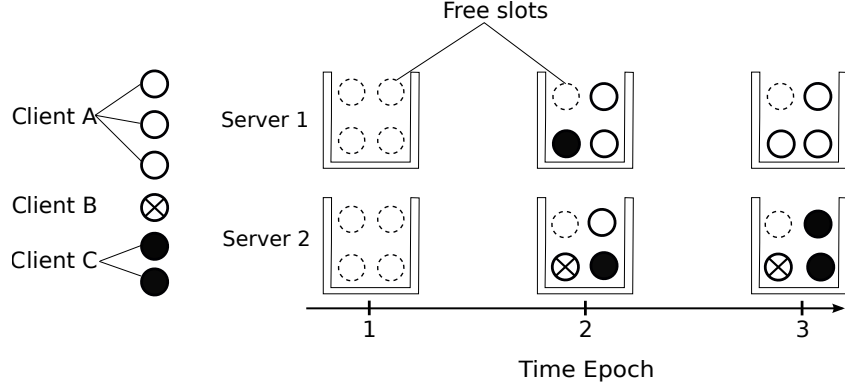
victim client’s VM to a malicious co-resident VM via a side-channel attack. In such an attack, the attacker runs a VM on the same server as that of the victim’s VM and takes advantage of a shared physical component in order to extract information about the victim. For example, the attacker may retrieve the victim’s cryptographic key by observing the activity of the processor cache. Prior work exposes several such co-residency side-channels in shared cloud environments (see, for example, [9, 10, 11, 12, 13, 14, 15, 16, 17]).

A straightforward solution to this security issue is hard-isolation: preclude co-residency and give each client dedicated hardware. This is unfavourable because it reduces efficient use of resources. Several other defences that do not involve the complete elimination of co-residency have been suggested by prior work (see, for example, [18, 19, 20]). Many of these defences suffer from at least one of the following drawbacks. Firstly, they entail changes to existing deployments and applications (see Section 3.6 and [5] for a discussion). Secondly, they are catered specifically to known cross-VM side-channel attacks. Therefore, as new attacker capabilities are unveiled, more changes may have to be made. It is desirable to have defences that are general across a broad spectrum of side-channel attacks and immediately deployable with no or only few modifications to existing cloud hardware and software.

With this objective in mind, scheduler-based defences have been proposed in prior work [5]. These can be thought of as a realization of the moving target defence philosophy to mitigate side-channels [21]. The mindset is that security is a factor which should be considered when deciding how VMs are placed on servers, rather than as an after-thought.

A scheduler has several mechanisms it can employ to maximize security. Two that have been explored in work by Moon et al. are the following [5]. A scheduler can be careful with where it places a newly arrived VM thereby limiting information leakage from or to it. Migration of existing VMs is yet another mechanism, with the intent that a victim VM does not leak too much information to a co-resident malicious VM. In Figure 3.1, we illustrate a scheduler-based defence. In the figure, a scheduler places and migrates client VMs in a manner that minimizes information leakage between them.

Scheduler-based defences appear to have promise as they offer several advantages. Firstly, they focus on the root cause of side-channels, i.e. co-residency, and are agnostic to the specific



**Figure 3.1.** Scheduling actions that can mitigate information leakage. Time is discretized into periods called *epochs*. A scheduler places VMs, such as the ones that arrive in Epoch 1 and are placed in Epoch 2. It also migrates VMs, which results in a new placement for Epoch 3. We use this example in several sections of the paper. In particular, the placement in Epoch 2 may or may not be considered to suffer from more information leakage than the one in Epoch 3, depending on the model for leakage that we adopt. ©2020 IEEE.

side-channel attack used. This makes them robust against unforeseen side-channels that meet certain conditions [5]. Secondly, they require no changes to the cloud provider’s hardware, client applications, and hypervisors and can be deployed “out of the box” as they require only changing the VM placement and/or scheduling algorithm deployed by the cloud provider. While we acknowledge that there may be other defences, e.g. shielded execution [22], that also offer these advantages to some extent, our focus is on scheduler-based defences.

An important contribution of Moon et al. [5] is four models that quantify information leakage in co-resident settings. These allow for the security-sensitive scheduling problem to be posed precisely as an optimization problem — given a migration budget, we seek a placement of VMs on servers that minimizes information leakage. That work explores three approaches to solve the problem, namely a mapping to ILP, an initial greedy algorithm called Baseline Greedy and a final greedy algorithm whose associated software is called Nomad.

Our primary intent is to analyze the overhead of introducing such security-sensitive scheduling

to cloud systems. We make three sets of contributions.

**Computational Complexity** We first pose and answer a fundamental question: is the underlying problem tractable<sup>2</sup>? A reason that this question is interesting is that we know that the cloud scheduling problem for other objectives, such as utilization, is intractable, i.e., **NP**-hard [23]. Is it the same for the objective of decreasing information leakage?

We first establish an upper-bound for the hardness of the problem. We show that a decision version of the problem, to which the optimization version reduces polynomially, is in **NP** (Section 3.3.1). We show also that the problem is indeed **NP**-hard and is therefore **NP**-complete (Section 3.3.2). We do this by showing that even for the simpler sub-case in which there are only two servers of equal capacity, deciding whether there exists a placement that meets a certain threshold of information leakage is **NP**-hard. A lower-bound hardness for that problem is a lower-bound for our more general problem for so-called open systems — settings in which VMs may come and go over time. We then establish that even the special case of the problem for closed systems, under the  $\langle R, C \rangle$  model for information-leakage, is **NP**-hard (Section 3.3.3). In closed systems, there already exists a placement of VMs, and there are no arrivals or departures. This special case is of interest because it is the only case supported by the implementation of Nomad that was made available to us.

Given the above results, and the customary assumption, that  $\mathbf{P} \neq \mathbf{NP}$ , no efficient (polynomial-time) algorithm exists for minimizing information leakage. We explore ways to mitigate this intractability (Section 3.3.4). In particular, we discuss our investigations into tractable sub-cases of the problem, and whether the problem may demonstrate certain kinds of optimal substructure.

**Analysis of Prior Approaches** Our second set of contributions is an analysis of the three approaches taken by Moon et al. to address security-sensitive scheduling [5].

---

<sup>2</sup>Some prior work uses the terms “tractable” and “intractable” to refer to an *approach* rather than the underlying *problem* for which the approach is proposed as a solution [5]. For clarity and in consistence with customary usage, we use the terms “tractable” and “intractable” for a problem, and “efficient” and “inefficient” for an approach.

That work asserts that mapping the problem to ILP is not promising in practice — even for modestly sized problem instances, the approach takes longer than a day for a commercial ILP solver on a reasonable computer. We analyze their mapping and observe that while it does appear to be a reduction, it can be inefficient<sup>3</sup> (Section 3.4.1) — there exists an encoding of placement for which the output of the mapping from that work is exponential in the size of its input. The fact that an underlying decision problem is in **NP** means that an efficient (polynomial-time computable) reduction to ILP exists. Because that work does not make use of an efficient reduction, it does not meaningfully characterize the overhead incurred by an ILP approach.

We then observe that their starting point in the design of a heuristic algorithm, Baseline Greedy, is, in the worst-case, more inefficient than it needs to be (Section 3.4.2). Given that a decision version of the underlying problem is in **NP**, we know that there exists an algorithm that can find the optimal solution in  $\Theta(c^n)$ , where  $n$  is the size of the input instance and  $c$  is a constant. An example of such an algorithm is one that tries every possible placement. We observe that baseline greedy runs in time  $\Theta(n!)$  in the worst-case. And  $c^n = o(n!)$ , that is, the function  $n!$  is an upper-bound for  $c^n$  that is not tight.

Their final design, Nomad, runs in time polynomial (quadratic) in the size of the input. Then, given that the underlying problem is **NP**-hard and the customary premise  $\mathbf{P} \neq \mathbf{NP}$ , we ask a natural question: what does Nomad give up for efficiency?

We show that Nomad provides no security for a large number of input instances (Section 3.4.3). Under the customary premise,  $\mathbf{P} \neq \mathbf{NP}$ , there are an infinite number of input instances for which Nomad provides no reduction in information leakage, let alone minimize it. Indeed, in the worst case, Nomad suffers the maximum information leakage possible, even when it is unconstrained by a migration budget, and even though there exists a placement that results in zero information leakage.

A natural consequent question one may ask is, can Nomad, or a similarly efficient algorithm, be modified to identify such instances and use a different approach on them, and yet retain its

---

<sup>3</sup>We distinguish correctness from efficiency in this context. A reduction is a mapping that satisfies a particular “if and only if” property. It may not be computable efficiently.

efficiency? The answer is ‘no,’ unless  $\mathbf{P} = \mathbf{NP}$  (Section 3.4.3).

**True Overhead** In our third set of contributions, we address the question as to what the overhead really is, for an approach that indeed minimizes information leakage. Our approach to answering the question is to first recognize that, as the problem is  $\mathbf{NP}$ -hard, we cannot hope to have an algorithm which is efficient for all inputs. However, a related decision-version is in  $\mathbf{NP}$ . Consequently, we have designed two approaches to the problem.

In one of our approaches, we reduce the problem to boolean satisfiability in conjunctive normal form (CNF-SAT) with the intent of adopting a SAT solver as an oracle. And in the other, we reduce the problem to ILP with the intent of adopting an ILP solver as an oracle. Both CNF-SAT and ILP are known to be  $\mathbf{NP}$ -complete [24]. Our reduction to ILP differs from that of prior work in that ours is computable in polynomial-time, and, as a consequence, the size of its output is polynomial in the size of the input. Our reductions to SAT and ILP are discussed in Section 3.5.

We have implemented both of our approaches, and we have conducted an empirical assessment of our approaches, and Nomad. We have also validated some of our analytical observations on Nomad against its implementation [1].

**Layout** We describe the problem we address in Section 3.2. This is the same problem that is addressed by Moon et al. [5]. Our results on the computational complexity of the problem are in Section 3.3. In Section 3.4, we describe and analyze the three approaches (i.e. a mapping to ILP, Baseline Greedy and Nomad) taken by Moon et al. [5]. We discuss our approaches that are based on reductions to CNF-SAT and ILP in Section 3.5. We address related work in Section 3.6 and conclude with Section 3.7.

We provide only a summary of the results from the empirical evaluation of our approaches and Nomad. For further details, the reader is referred to our paper [1], and other work [8].



## 3.2 Minimizing Information Leakage

In this section, we describe the problem posed by Moon et al. [5]. The problem is: given (i) a set of clients, each having a number of VMs, (ii) a set of servers, each having a capacity expressed as a number of VMs, (iii) a migration budget, that limits the number of VM migrations that can occur, and (iv) previous placements of VMs on servers, what is a new placement that minimizes information leakage? Our exposition is similar to that of Moon et al. [5], with changes for clarity only.

**Setup** Time is discretized into periods called *epochs*. A *VM* is a virtual machine — a unit of execution, similar to a process in a conventional operating system. A VM, when run, may last several epochs. A *client* is associated with a set of VMs; every VM belongs to a client. A VM belongs to the same client for the duration of its lifetime. The set of VMs that belong to a client can change over time. A *server* is a machine on which a VM runs; the VM is said to be *hosted* by that server. A server is associated with a capacity — it is able to host a certain number of VMs only, at any given time. VMs may be *migrated* — hosted by a different server in a later epoch. A potential victim client has some secret information in her VMs. A malicious client attempts to steal the secret of a victim client via information-leakage across VMs that can occur when the VMs are *co-resident*, i.e., hosted by the same server. We do not know beforehand which VMs are potential victims, and which are malicious. In the leakage models we consider, clients do not share secrets, and a client’s secret may be present in more than one of its VMs. Figure 3.1 shows three servers and clients, and six VMs across those three clients, and hosting over three epochs.

We quantify information as a certain number of bits. We premise that merely owing to co-residency of a victim and a malicious VM, information leaks from the former to the latter. Information that leaks from a VM to another aggregates over the time that the two VMs are co-resident. In order to aggregate information leakage as a function of co-residency over time, we consider a *sliding window* of the most recent  $\Delta$  epochs. This can model secrets, such as cryptographic keys, that are refreshed periodically; information the adversary gathered in previous sliding windows is no longer useful to her. We assume that during an epoch, a VM leaks 1 bit to another that is co-resident. This can be generalized to the case that the amount of

leakage is some constant,  $k$  bits. The value  $k$  can then be used to model the time-length of an epoch. The amount of information leaked from a victim to an adversary is proportional to the time their VMs are co-resident with one another. For example, if a VM of Client A is co-resident with a VM of Client B for 2 epochs, then the total information leaked from Client A's VM to Client B's VM over the 2 epochs is 2 bits.

**Replication and Collusion** Replication means that information is duplicated across a victim client's VMs. This is potentially advantageous for an attacker because when an attacker VM is co-resident with multiple victim VMs, under the assumption that the attacker VM extracts different bits of information from each victim VM, the rate of leakage to the attacker increases. For example, in Figure 3.1, for the placement in Epoch 2, with replication, Client C (black), on Server 1, gains twice as many bits of a secret from Client A (white) than it would without replication.

Collusion means that a malicious client's VMs can collaborate with one another. This is potentially advantageous for the attacker because when multiple attacker VMs are co-resident with a victim VM, under the assumption that each attacker VM extracts different bits from the victim VM, the rate of leakage to the attacker, as a whole, increases. In Figure 3.1, for the placement in Epoch 2, with collusion, Client A (white), on server 1, gains twice as many bits of a secret from Client C (black) than it would without. As in prior work, we denote the presence of replication and collusion by  $R$  and  $C$  respectively, and the absence by  $NR$  and  $NC$  respectively. This leads to four possible leakage models:  $\langle NR, NC \rangle$ ,  $\langle R, NC \rangle$ ,  $\langle NR, C \rangle$ ,  $\langle R, C \rangle$ .

Table 3.1 presents the symbols we use in our calculations of information-leakage.  $v_{c,i}$  denotes the  $i^{\text{th}}$  VM of client  $c$ . The VM-to-VM information leakage between the VMs  $v_{c,i}$  and  $v_{c',i'}$  over the most recent  $\Delta$  epochs, denoted by  $I_{c,i,c',i'}(e, \Delta)$ , is calculated as the number of epochs out of the  $\Delta$  epochs that the two VMs were co-resident with each other. The VM-to-VM leakages between VMs of clients  $c$  and  $c'$  are used to calculate the client-to-client leakage between clients  $c$  and  $c'$ , denoted by  $I_{c,c'}(e, \Delta)$ . The manner in which this is done depends on

Symbol	Meaning
$v_{c,i}$	The $i^{\text{th}}$ VM of client $c$ .
$Y_{c,i,c',i'}(e')$	An indicator variable, i.e., it takes the value 0 or 1 only. It takes the value of 1 if $v_{c,i}$ and $v_{c',i'}$ are co-resident on a server during an epoch $e'$ . Otherwise it takes the value 0.
$I_{c,i,c',i'}(e, \Delta)$	The VM-to-VM information leakage from $v_{c,i}$ to $v_{c',i'}$ over the sliding window of epochs $(e - \Delta, e]$ . This quantity is calculated as $\sum_{e' \in (e - \Delta, e]} Y_{c,i,c',i'}(e')$
$I_{c,c'}(e, \Delta)$	The client-to-client information leakage from $c$ to $c'$ over the sliding window of epochs $(e - \Delta, e]$ . This quantity is dependent on the presence or absence of replication and collusion.
$I_{total}(e, \Delta)$	The total information leakage over the sliding window of epochs $(e - \Delta, e]$ , under one of the four leakage models, $\{R, NR\} \times \{C, NC\}$ . This quantity is calculated by aggregating the client-to-client leakage across all pairs of clients.

**Table 3.1.** The symbols used for calculating total information leakage. ©2020 IEEE.

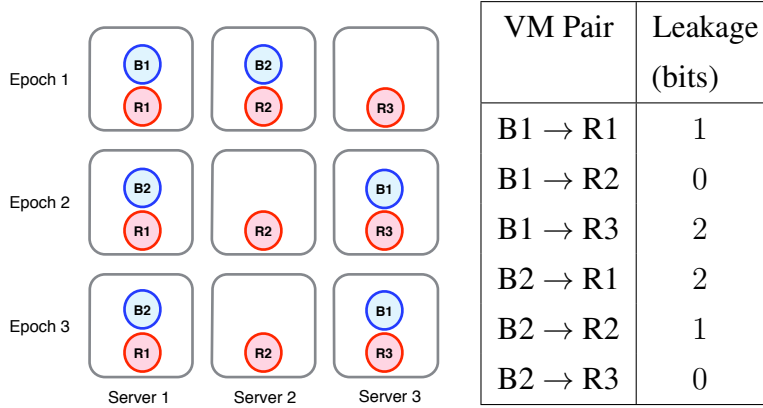
the leakage model adopted.

**Information Leakage Models** We now describe how client-to-client leakage is calculated under the four models, using the example in Figure 3.2.

Model 1:  $\langle NR, NC \rangle$  In this model there is neither replication nor collusion. The information leakage from client  $c$  to client  $c'$  is the maximum per-VM-pair information leakage across all pairs of VMs. This is the information which leaks to the adversary VM that is co-resident with the same victim VM for the largest number of epochs:

$$I_{c,c'}(e, \Delta) = \max_i \max_{i'} I_{c,i,c',i'}(e, \Delta) \quad (3.1)$$

*Example:* In Figure 3.2, the maximum VM-to-VM leakage from the blue client (B) to the red client (R) is 2 bits, so the leakage from the blue client to the red client under  $\langle NR, NC \rangle$  is 2



**Figure 3.2.** Example used in the prose to show how client-to-client leakage is calculated under all four models. The leakage from B1 to R1 is 1 bit over the three epochs because B1 is co-resident with R1 for one out of the three epochs.

bits over the three epochs.

Model 2:  $\langle R, NC \rangle$  In this model there is replication but no collusion. There is a cumulative effect across victim VMs. The information leakage from client  $c$  to client  $c'$  is determined by the adversary VM that has extracted the most information:

$$I_{c,c'}(e, \Delta) = \max_{i'} \left[ \sum_i I_{c,i,c',i'}(e, \Delta) \right] \quad (3.2)$$

*Example:* In Figure 3.2, R1 obtains the most information, a total of  $1+2=3$  bits, so the leakage from the blue client to the red client under  $\langle R, NC \rangle$  is 3 bits.

Model 3:  $\langle NR, C \rangle$  In this model there is collusion but no replication. There is a cumulative effect across adversary VMs. The information leakage from client  $c$  to client  $c'$  is determined by the victim VM that has leaked the most information:

$$I_{c,c'}(e, \Delta) = \max_i \left[ \sum_{i'} I_{c,i,c',i'}(e, \Delta) \right] \quad (3.3)$$

*Example:* In Figure 3.2, B1 and B2 each leak a total of  $1+2=3$  bits, so the leakage from the blue client to the red client under  $\langle NR, C \rangle$  is 3 bits.

**Model 4:  $\langle R, C \rangle$**  In this model there is both replication and collusion. There are cumulative effects across both victim and adversary VMs:

$$I_{c,c'}(e, \Delta) = \sum_i \sum_{i'} I_{c,i,c',i'}(e, \Delta) \quad (3.4)$$

*Example:* In Figure 3.2, the sum of all bits leaked from the blue VMs to red VMs is 6, so the leakage from the blue client to the red client under  $\langle R, C \rangle$  is 6 bits.

**Total Information Leakage** Given the client-to-client information leakage in any one of the four models, the total information leakage that we seek to minimize is:

$$I_{total}(e, \Delta) = \sum_c \sum_{c'} I_{c,c'}(e, \Delta) \quad (3.5)$$

Minimizing  $I_{total}(e, \Delta)$  is equivalent to minimizing the average leakage across client pairs. We acknowledge that there may be other ways to represent the security of the system, for example, by minimizing the inter-client leakage. We chose the above representation for ease of comparison with prior work [5]. However the approaches we propose in Section 3.5 are general and can be used for other representations of security by modifying the objective function.

An observation is that the total information leakages computed under the  $\langle NR, C \rangle$  and  $\langle R, NC \rangle$  models are equal.

**Problem Statement** We now pose the scheduling problem that is solved at the beginning of each epoch with the intent of minimizing information-leakage. We first choose and fix a particular information leakage model from amongst  $\{R, NR\} \times \{C, NC\}$ .

Inputs: The lists  $X, S, A, F$ , and an integer,  $m$ .  $X = \{x_1, x_2, \dots, x_n\}$  is a set of integers where  $x_i$  denotes the number of VMs belonging to the  $i^{\text{th}}$  client. The number of clients,  $n$ , is expressed via the size of this set.  $S = \{s_1, s_2, \dots, s_k\}$  is a set of integers where  $s_j$  denotes the number of VMs the  $j^{\text{th}}$  server can host. The number of servers,  $k$ , is expressed via the size of this set.  $A = \{a_1, a_2, \dots, a_n\}$  is a set of integers where  $a_i$  denotes the number of

Symbol	Meaning
$X, x_i$	$X = \{x_1, x_2, \dots, x_n\}$ is a list of integers where $x_i$ denotes the number of VMs that belong to the $i^{\text{th}}$ client.
$S, s_j$	$S = \{s_1, s_2, \dots, s_k\}$ is a list of integers where $s_j$ denotes the number of VMs the $j^{\text{th}}$ server can host.
$A, a_i$	$A = \{a_1, a_2, \dots, a_n\}$ is a list of integers where $a_i$ denotes the number of VMs belonging to the $i^{\text{th}}$ client which have arrived in this epoch.
$F, f_l$	$F = \{f_{t-1}, f_{t-2}, \dots, f_{t-\Delta+1}\}$ , where each $f_l$ encodes the placement of VMs in servers in the $l^{\text{th}}$ epoch. Entries in $F$ can be used to express VMs that depart.
$m$	An integer that is the migration budget. Each kind of migration (e.g., swap VMs on two servers) has a cost associated with it. A special symbol, $\infty$ , for $m$ , is used to indicate that we are to be unconstrained by a migration budget.

**Table 3.2.** The inputs to the scheduling problem. ©2020 IEEE.

VMs belonging to the  $i^{\text{th}}$  client which have arrived just before the current epoch.  $m$  is an integer that is the migration budget. Each kind of migration has a cost associated with it. For example, moving a VM from one server to another may cost 1 unit of migration. Swapping a VM on one server with a VM on another costs double of this, i.e., 2 units of migration.  $F = \{f_{t-1}, f_{t-2}, \dots, f_{t-\Delta+1}\}$  is the set of placements that were used in prior epochs, where each  $f_l$  encodes the assignment of VMs to servers in the  $l^{\text{th}}$  epoch. Entries in  $F$  can be used to express VMs that depart. Table 3.2 summarizes these inputs.

Output: A placement for the current epoch,  $f_t$ , that corresponds to the minimum total information leakage subject to the constraints specified as input.

As an example, suppose that the input is as shown for Epoch 2 in Figure 3.1. Assume the leakage model is  $\langle R, C \rangle$ , and  $m = 2$ . Also assume that Client A corresponds to Client 1, Client B to Client 2 and so on. Then, the inputs are:  $X = \{3, 1, 2\}$ ,  $S = \{4, 4\}$ ,  $A = \{0, 0, 0\}$  and a function  $f_2(i, j)$  which indicates the number of VMs of client  $i$  on server  $j$  during Epoch 2, e.g.  $f_2(1, 1) = 2$ . The output is the placement shown for Epoch 3 in Figure 3.1. In this

example, the total information leakage in Epoch 2 is 10 bits, and in Epoch 3 it is 14 bits (10+4). That is, migrating some VMs causes the increase in information-leakage to be 4 only, rather than 10, which would have been the case had we adopted the same placement for Epoch 3 as we have for Epoch 2.

### 3.3 Computational Complexity

We ask: does there exist an efficient algorithm for the problem from the previous section? If the problem is intractable (e.g., **NP**-hard), then the answer is, ‘no’, under customary assumptions (e.g.,  $\mathbf{P} \neq \mathbf{NP}$ ). In this section, we identify the computational complexity of the problem. In Section 3.3.1 we present an upper-bound on the complexity: a decision problem that is related polynomially is in **NP**. In Section 3.3.2 we present a lower bound: the problem is **NP**-hard. In Section 3.3.3, we establish that the special case that is implemented in the version of Nomad we have been provided is **NP**-hard as well. In Section 3.3.4 we discuss ways in which the hardness can be mitigated, and some interesting problems that remain open.

#### 3.3.1 An Upper-Bound: **NP**

Consider the decision version of the secure scheduling problem in which we are given a threshold information leakage  $q$ , and we ask if there exists a placement for the next epoch such that all the input constraints are satisfied and the total information leakage incurred is at most  $q$ . This problem is in **NP**. An efficient (polynomially sized) certificate is a placement for the next epoch. We can verify the certificate against all the constraints in polynomial time.

Given a placement for the next epoch, we can use it together with the input  $F$ , which is a set of placements used in the previous  $\Delta - 1$  epochs, to efficiently compute the information leakage it incurs.  $\Delta$  is the number of epochs in a sliding window. For every client pair, for every VM pair of those clients, we compute the VM-to-VM leakage as the number of epochs from the  $\Delta$  epochs that the VM-pair have been co-resident with each other. Then we compute

client-to-client information leakage using the appropriate formula from Equations 3.1-3.4 depending on the model of information leakage adopted. Finally, we aggregate client-to-client information leakage over all pairs of clients as in Equation 3.5. Hence we can efficiently check whether the total information leakage incurred by the certificate placement is less than or equal to the threshold information leakage  $q$ .

We can also efficiently check whether the placement respects server capacities and assigns each VM to exactly one server. Finally, we can compare this output placement to the placement of the immediately prior epoch to determine the number of migrations needed, and compare this to the budget  $m$ .

The optimization problem posed in Section 3.2 can be reduced in polynomial time to the above decision problem by doing a binary search on the threshold to find the minimum information leakage. Hence given that the decision version is in **NP**, an approach to solve the optimization problem is to use an oracle for the decision version. We discuss ILP and CNF-SAT solvers as oracles in Section 3.5.

### 3.3.2 A Lower-Bound: NP-hard

To prove that the problem is **NP**-hard, we show that a problem which is known to be **NP**-hard can be reduced to it. We pose a variant of the classical Set Partition problem, which is known to be **NP**-hard [25]. We call this variant *Equal Cardinality Partition* and show that it is **NP**-hard as well. For the  $\langle NR, NC \rangle$  model, we show an efficient reduction from Equal Cardinality Partition. For the other three models, we show an efficient reduction from Set Partition. For all four leakage models, we reduce to a sub-case of the decision version of our problem, that we call Initial Placement, thereby proving that the general problem is **NP**-hard.

We begin with defining the Set Partition problem. In Set Partition, we are given a multiset  $S = \{x_1, x_2, \dots, x_n\}$  of positive integers, and we seek to partition  $S$  into two subsets  $S_1$  and  $S_2$  such that the sum of the integers in  $S_1$  equals the sum of the integers in  $S_2$ . If indeed this is the case for a particular instance of the problem, we say that the instance is true and call the sets  $S_1, S_2$  a solution to the instance. Otherwise we say that the instance is false.



In Equal Cardinality Partition, we impose the additional constraint that  $|S_1| = |S_2|$ . We also require that in every input instance, both the number of integers,  $n$ , as well as the sum of the integers,  $\sum_i x_i$ , are even, since otherwise the instance has no solution. We can check these two conditions efficiently and reject any input that does not meet them.

**Lemma 3.3.1.** *Equal Cardinality Partition is NP-hard.*

*Proof.* Our mapping from Set Partition to Equal Cardinality Partition is as follows. Given an instance of Set Partition,  $S = \{x_1, x_2, \dots, x_n\}$ , increment each integer in  $S$  by 1, and then add  $n$  1's to  $S$ . Output the resultant set,  $S' = \{x_1 + 1, x_2 + 1, \dots, x_n + 1, 1_1, 1_2, \dots, 1_n\}$ , as the corresponding instance of Equal Cardinality Partition. This mapping is computable efficiently; in linear time. It is a reduction if, given an instance of Set Partition, it is true if and only if the instance of Equal Cardinality Partition output by the mapping is true. Assume that the input Set Partition instance is true. Therefore, there exists a solution, two subsets  $S_1, S_2$  whose contents sum to the same. If we increment every element in  $S_1$  and  $S_2$  by 1, add  $|S_1|$  1's to  $S_2$  and  $|S_2|$  1's to  $S_1$ , we obtain a solution to the corresponding Equal Cardinality Partition instance. That is, the instance of Equal Cardinality Partition produced by the mapping is true. For the other direction, assume that the Equal Cardinality Partition instance output by the mapping for some input instance of Set Partition is true. Then, it has a solution  $S_1$  and  $S_2$ . We make the following observation. There are  $2n$  integers in  $S'$ :  $n$  of them are 1, and  $n$  of them are not 1, and hence  $|S_1| = |S_2| = n$ . If  $S_1$  has  $k$  1's, then  $S_1$  must have  $n - k$  integers that are not 1's, and it follows that  $S_2$  must have  $n - k$  1's and  $k$  integers that are not 1's. If we remove all the integers equal to 1 from  $S_1$  and  $S_2$ , and decrement every remaining element in each of the sets by 1, we obtain two sets that demonstrate that the input instance of Set Partition is true.  $\square$

Now consider the following sub-case of the problem we pose in Section 3.2. We are given a list  $X = \{x_1, \dots, x_n\}$  of client VMs, where  $n > 1$  and  $\sum_i x_i$  is even, and a list  $S = \{s_1, s_2\}$  of two servers only, with  $s_1 = s_2 = (\sum_i x_i) / 2$ . We can infer  $S$  from  $X$ ; that is,  $S$  does not have to be explicitly specified in the input for this sub-case of the general problem. The other inputs are  $A = X$ ,  $F = \emptyset$ , and  $m = \infty$ . That is, we are asked to place the VMs from  $X$  across two equal-capacity servers in a way that minimizes information leakage. The capacity

of each server is exactly half the total number of VMs that need to be placed. We show that this sub-case which we call the Initial Placement problem, is **NP**-hard, thereby proving that the general problem is as well.

The above version of the Initial Placement problem is an optimization problem. To show that it is **NP**-hard, we consider the following decision version of it. It takes all the same inputs, i.e.,  $X, A, F$  and  $m$ , and an additional input,  $q$ , an integer that is a threshold for the information leakage. The instance is true if there exists a placement which results in an information leakage of at most  $q$  bits, and false otherwise. The decision version can be reduced to the optimization version. Given an oracle for the optimization version, we simply query it with the inputs  $X, A, F$  and  $m$ , and compare its output to  $q$ . Thus, if this decision version is **NP**-hard, so is the optimization version.

Before we show that Initial Placement is **NP**-hard, we state and prove Lemma 3.3.2 which is used in the reductions.

**Lemma 3.3.2.** *For any instance of Initial Placement, a lower-bound on the minimum information leakage for the  $\langle NR, NC \rangle$  model is the one that arises when all the VMs of exactly half the clients are on one server, and all the VMs of the other half of the clients are on the other server. And this is the only placement that achieves this lower-bound. For the other three models, a lower-bound on the minimum information leakage is achieved when each client has all of its VMs on the same server. And this is the only placement that achieves this lower-bound.*

*Proof.* We begin with the  $\langle NR, NC \rangle$  model. The information leakage resulting from placing the VMs of half the clients on the first server, and the VMs of the other half of the clients on the other server is  $2 \times 2 \times \binom{n/2}{2}$ . The “ $\binom{n/2}{2}$ ” term arises from having  $n/2$  clients on each server. It gives us the number of pairs of clients on each server. One of the terms “2” in the expression arises from considering both pairs  $\langle c, c' \rangle$  and  $\langle c', c \rangle$  of clients. We do this as information leaks in both directions. The other “2” term arises from having two servers.

There are only two other possible kinds of placements. In the first, one of the two servers hosts VMs belonging to more than  $n/2$  clients, and the other server hosts VMs belonging to at

least  $n/2$  clients. In this case the proof follows because for  $k_1 \geq 0$  and  $k_2 \geq 1$ , the following inequality holds:

$$\binom{n/2}{2} + \binom{n/2}{2} < \binom{n/2 + k_1}{2} + \binom{n/2 + k_2}{2} \quad (3.6)$$

In the second kind of placement, exactly one of the servers is hosting VMs belonging to less than  $n/2$  clients. In this case, the proof follows because for  $1 \leq k_1 \leq n/2$ ,  $1 \leq k_2 \leq n/2$ ,  $n/2 - k_1 + n/2 + k_2 \geq n$ , the following inequality holds:

$$\binom{n/2}{2} + \binom{n/2}{2} < \binom{n/2 - k_1}{2} + \binom{n/2 + k_2}{2} \quad (3.7)$$

For the  $\langle NR, C \rangle$  model, observe that if each client has all of its VMs on the same server, then if  $T$  denotes the total number of VMs, the total information leakage is given by:

$$\sum_{i \in [1, n]} (T/2 - x_i) \quad (3.8)$$

Consider a placement in which not every client has all of its VMs on the same server. Let the number of VMs of client  $i$  on server 1 and server 2 be  $a_i$  and  $b_i$  respectively. Let  $M_{ij}$  be the maximum number of client  $j$ 's VMs that are co-resident with the same VM of client  $i$ . Then the total information leakage is given by:

$$\sum_{i \in [1, n]} \sum_{\substack{j \in [1, n] \\ j \neq i}} M_{ij} \quad (3.9)$$

Now observe that for a given client  $i$ :

$$\sum_{\substack{j \in [1, n] \\ j \neq i}} M_{ij} \geq T/2 - \max\{a_i, b_i\} \quad (3.10)$$

To see why Equation 3.10 is true, assume for the purpose of contradiction the opposite. Also assume, without loss of generality, that  $a_i \geq b_i$ . Then we have:

$$\sum_{\substack{j \in [1, n] \\ j \neq i}} M_{ij} < T/2 - a_i \quad (3.11)$$

If  $a_i = x_i$  and  $b_i = 0$ , then:

$$\sum_{\substack{j \in [1, n] \\ j \neq i}} M_{ij} < T/2 - x_i \quad (3.12)$$

which is false, because  $\sum_{j \neq i} M_{ij}$  is the total leakage from client  $i$ , and if all of client  $i$ 's VMs are on the same server, we know that the total leakage from client  $i$  to all other clients is exactly:

$$\sum_{\substack{j \in [1, n] \\ j \neq i}} M_{ij} = T/2 - x_i \quad (3.13)$$

Now consider the other case in which  $a_i < x_i$  and  $b_i > 0$ . Then  $M_{ij}$  for  $j \neq i$ , is the maximum number of VMs that client  $j$  has on a server. Let  $L_j$  be the minimum number of VMs that client  $j$  has on a server. Then for all  $j \neq i$ :

$$L_j \leq M_{ij} \quad (3.14)$$

We also know that:

$$\sum_{\substack{j \in [1, n] \\ j \neq i}} L_j + \sum_{\substack{j \in [1, n] \\ j \neq i}} M_{ij} + x_i = T \quad (3.15)$$

If Equation 3.11 holds, then this means:

$$\sum_{\substack{j \in [1, n] \\ j \neq i}} L_j > T - x_i - (T/2 - a_i) = T/2 - b_i \geq T/2 - a_i > \sum_{\substack{j \in [1, n] \\ j \neq i}} M_{ij} \quad (3.16)$$

This contradicts Equation 3.14 which implies Equation 3.11 is false. Having now established that Equation 3.10 holds, we combine Equations 3.9 and 3.10 to get the total information leakage, for a placement that does not place all the VMs of each client on the same server, to be the following, thereby proving Lemma 3.3.2 for the  $\langle NR, C \rangle$  model:

$$\sum_{i \in [1, n]} \sum_{\substack{j \in [1, n] \\ j \neq i}} M_{ij} \geq \sum_{i \in [1, n]} T/2 - \max\{a_i, b_i\} \geq \sum_{i \in [1, n]} T/2 - x_i \quad (3.17)$$

For the  $\langle R, NC \rangle$  model, we first observe that the model is symmetric to the  $\langle NR, C \rangle$  model. That is, the leakage from client  $c$  to client  $c'$  under  $\langle R, NC \rangle$  is equal to the leakage from  $c'$  to  $c$  under  $\langle NR, C \rangle$  (see Equations 3.2 and 3.3 in Section 3.2). Hence the total information leakage under both models are equal and it follows that Lemma 3.3.2 holds for the  $\langle R, NC \rangle$  model.

In the  $\langle R, C \rangle$  model, a placement for an Initial Placement instance that places all the VMs of each client on the same server results in total information leakage of

$$\sum_{i \in [1, n]} x_i \times (T/2 - x_i) \quad (3.18)$$

The total information leakage for a placement that does not have this property is given by the expression below, in which  $a_i + b_i = x_i$ :

$$\sum_{i \in [1, n]} a_i \times (T/2 - a_i) + b_i \times (T/2 - b_i) \quad (3.19)$$

□

The expression in Equation 3.19 is greater than that in Equation 3.18. Thus Lemma 3.3.2 holds for the  $\langle R, C \rangle$  model.

**Theorem 3.3.3.** *Initial Placement is NP-hard.*

*Proof.* For the  $\langle NR, NC \rangle$  model, the proof is a reduction from Equal Cardinality Partition. Given an instance,  $S = \{x_1, x_2, \dots, x_n\}$ , of Equal Cardinality Partition, let  $X = \{x_1, x_2, \dots, x_n\}$ , and  $s_1 = s_2 = (\sum_i x_i) / 2$ . We set  $q$  to be the lower-bound on the minimum information leakage described in Lemma 3.3.2. That is, we set  $q = 2 \times 2 \times \binom{n/2}{2}$ . This reduction is computable efficiently, in linear time. We assert that the instance of Equal Cardinality Partition is true if and only if the corresponding instance of Initial Placement is true.

Assume that the input Equal Cardinality Partition instance is true. Therefore, there exists a solution, two equal sized subsets,  $S_1$  and  $S_2$ , whose contents sum to the same. These two subsets give us the placement across the two servers which results in an information leakage of  $q$  for the resulting Initial Placement instance under the mapping. This follows directly from Lemma 3.3.2. Hence, the instance of Initial Placement produced by the mapping is true. For the other direction, assume that the instance of Initial Placement output by the mapping for some input instance of Equal Cardinality Partition is true. By Lemma 3.3.2, we know that this means there exists a placement such that all the VMs of exactly half of the clients are on one server, and all the VMs of the other half of the clients are on the other server. Then a solution to the Equal Cardinality instance is  $S_1 = \{x_i \in X : \text{client } i \text{ is placed on server 1}\}$  and  $S_2 = \{x_i \in X : \text{client } i \text{ is placed on server 2}\}$ . By Lemma 3.3.2 the cardinalities of the two sets are equal. Since each server has a capacity equalling half the total number of VMs, the contents of  $S_1$  and  $S_2$  sum to the same.

For the other three models, the proof is a reduction from the version of Set Partition in which the sum of all integers in  $S$  is even, to Initial Placement. This version of Set Partition is NP-hard because any instance of Set Partition whose sum of integers is odd must be a false instance. Given an instance,  $S = \{x_1, x_2, \dots, x_n\}$ , of Set Partition, let  $X = \{x_1, x_2, \dots, x_n\}$ , and  $s_1 = s_2 = (\sum_i x_i) / 2$ . We set  $q$  to be the lower bound on the minimum information leakage described in Lemma 3.3.2. For the  $\langle R, C \rangle$  model, this lower bound is  $\sum_i (T/2 - x_i) \times x_i$ , where  $T$  is the total number of VMs. The other two models are symmetric and the lower bound for both is  $\sum_i (T/2 - x_i)$ . The reduction can be computed efficiently. We assert that the

input Set Partition instance is true if and only if the resulting Initial Placement instance is true, using reasoning similar to that used to demonstrate correctness for the reduction from Equal Cardinality Partition to Initial Placement in the  $\langle NR, NC \rangle$  model.  $\square$

The Initial Placement problem is a sub-problem of the general problem that is posed in Section 3.2. From Theorem 3.3.3, it follows that the general problem, i.e., for any  $k \geq 2$  servers, is also **NP**-hard.

### 3.3.3 Closed Migration under the $\langle R, C \rangle$ Model

We have shown in the previous section that the problem is **NP**-hard in general. In this section, we consider a particular special case of the problem: closed migration, under the  $\langle R, C \rangle$  model for information-leakage. With closed-migration, there is a set of existing VMs, and no VMs arrive or leave. A reason we consider this special case in particular is that the implementation of Nomad we were provided supports this special case only. Consequently, for the approaches based on reduction to ILP and CNF-SAT that we discuss in Section 3.5 and have implemented to compare empirically with Nomad [1], this special case is of particular interest.

We observe that under the  $\langle R, C \rangle$  model for information leakage, a placement,  $f$ , needs to express the number of a client's VMs that are placed on a server only, and not the particular VMs that are placed on the server. That is,  $f$  can be perceived as a function,  $f: [1, n] \times [1, k] \rightarrow [0, \max_i \{x_i\}]$ , and can be encoded as a table of size  $n \times k \times \log \max_i \{x_i\}$  bits. This is possible because the formula for inter-client information leakage does not contain any maximization (i.e., “max”) terms (see Section 3.2).

It is easy to compute a placement that expresses which specific client-VMs are placed on a server given such a more compact encoding for placement. Following is an algorithm. Let the new placement be  $f_t$  and the immediately prior placement be  $f_{t-1}$ . Let  $x_{ij} = f_t(i, j) - f_{t-1}(i, j)$ . Place all newly arriving VMs in a pool. For all  $i \in [1, n]$  (i.e., the clients) and  $j \in [1, k]$  (i.e., the servers), if  $x_{ij} < 0$ , remove  $|x_{ij}|$  VMs of client  $i$  from server  $j$  and place them in the pool. Then, for all  $i \in [1, n]$  and  $j \in [1, k]$ , if  $x_{ij} > 0$ , take  $x_{ij}$  VMs of client  $i$

from the pool and place them on server  $j$ . This algorithm guarantees the fewest migrations.

Computing such a placement after we have solved a problem-instance is more efficient: the input to and output from an algorithm for the optimization problem, and a corresponding decision problem, are more compact. We exploit this observation regarding encoding for the  $\langle R, C \rangle$  model to assert the following theorem.

**Theorem 3.3.4.** *Closed Migration under  $\langle R, C \rangle$  is **NP**-hard.*

*Proof.* We show that a decision version of a sub-case of the problem is **NP**-hard. This establishes that the more general problem is also **NP**-hard. The sub-case is as follows. We are given as input  $X = \{x_1, x_2, \dots, x_n\}$ , i.e., VMs that correspond to clients. We are given  $S = \{s_1, s_2\}$  with  $s_1 = s_2 = (\sum_i x_i) / 2$ , i.e., two servers only each with capacity exactly half of the total number of VMs. And we are given as input  $F = \{f_0\}$ , where  $f_0$  is some placement of the VMs across the two servers. Also,  $A = \emptyset$  and  $m = \infty$ . Suppose every placement  $f$  is encoded as a function that, given a client and server, maps it to the number of VMs of that client that are resident on the server. That is,  $f: [1, n] \times \{1, 2\} \rightarrow [0, \max_i \{x_i\}]$ . The decision version, in addition to these inputs  $X, S$  and  $F$ , takes an integer  $q'$ . We ask whether there exists a new placement,  $f_1$ , such that the total information leakage across the two epochs is  $q'$ .

We reduce from the Initial Placement problem (see Section 3.3.2). Suppose, in the input instance of the Initial Placement problem, we are given  $X$  and the information leakage threshold,  $q$ . We infer  $S$  from  $X$ , and adopt the same  $X$  and  $S$  in our output instance of the Closed Migration problem. Also, for the initial placement of the Closed Migration problem,  $f_0$ , we use some straightforward efficient algorithm to apportion VMs across the two servers. For example, assign  $s_1$  VMs to the first server, where  $s_1$  is the capacity of the server, then assign the remaining VMs to the second server. For  $q'$ , the information leakage threshold for the output Closed Migration instance, we simply adopt the sum of the leakage caused by the placement  $f_0$  (which we can efficiently compute), and the threshold,  $q$ , from the input Initial Placement instance, as we show in the formula below. The double summation of products of the  $f_0$  values arises from the fact that we are under the  $\langle R, C \rangle$  model for information leakage.



$$q' = q + \sum_{\substack{c, c' \in [1, n] \\ c' \neq c}} \sum_{j \in [1, k]} f_0(c, j) \times f_0(c', j)$$

We assert that the Initial Placement instance is true if and only if the corresponding Closed Migration instance is true.  $\square$

### 3.3.4 Mitigating NP-hardness and Open Problems

We now discuss possible ways in which the NP-hardness of the scheduling problem we consider can be mitigated, and open problems that remain in this context.

Our intractability results and the customary assumption that  $\mathbf{P} \neq \mathbf{NP}$  imply that no efficient algorithm exists that can find an optimal placement for every input instance. Possible ways to deal with the intractability are: (1) Identify sub-cases that are tractable. (2) Check if the problem exhibits optimal substructure in order to determine if dynamic programming or a greedy approach can be used. (3) Seek efficient approximability. We discuss our investigations into (1) and (2). (3) is topic for future work.

Nomad [5] falls under (1), in that it achieves optimality for some sub-cases via an efficient algorithm. However, in that work, exactly which sub-cases are tractable, i.e., can be solved efficiently, is not identified. In Theorem 3.4.3 in Section 3.4.3 we show that Nomad is ineffective for infinitely many classes of input, each with infinitely many inputs, unless  $\mathbf{P} = \mathbf{NP}$ .

**Tractable Sub-cases** We have identified two tractable sub-cases. Consider the sub-case in which every client  $i$  has only one VM,  $v_i$ . An efficient algorithm to compute the optimal placement for this sub-case is to spread the VMs evenly across servers. When a server is full, it is eliminated from consideration. Once the optimal placement has been determined, the migration budget over the following epochs can be used to get from the current placement to the optimal.

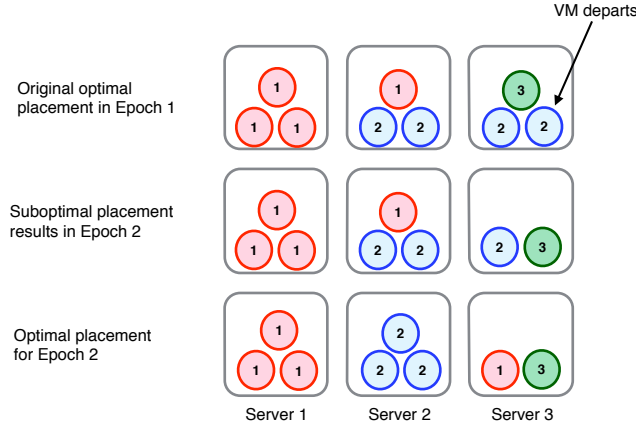
Another tractable sub-case is one in which the minimum information leakage that can be

achieved is zero. This is possible if and only if for every client  $i$  there exists a designated server  $j$  such that the server's capacity is at least the number of VMs client  $i$  has, i.e.,  $s_j \geq x_i$ . There exists an efficient algorithm to check for this condition.

**Optimal Substructure** A problem is said to exhibit optimal substructure if, given an optimal solution for an instance, removal of some components from both the problem instance and the solution, yields an optimal solution for the smaller problem instance. Examples of approaches that exploit optimal substructure are greedy algorithms and dynamic programming. Note that we do not expect a problem that is strongly **NP**-hard to exhibit optimal substructure, as this would imply that **P** = **NP**. However, if a problem is weakly **NP**-hard, then it may. A problem is said to be weakly **NP**-hard only, if there exists an encoding of its inputs (e.g., in unary) for which it is tractable. Some versions of the well-known Knapsack problem, for example, are only weakly **NP**-hard. As we do not know whether some versions of our problem are weakly **NP**-hard, our investigations into whether the problem exhibits optimal substructure have implications to whether the problem is only weakly **NP**-hard.

Consider the general problem. Let  $\pi$  be an optimal placement for some input instance. Pick a VM  $v$  that belongs to some client  $i$ . Suppose that in  $\pi$ ,  $v$  is placed on server  $s$ . Remove  $v$ , decrement the capacity of  $s$  by 1 and decrement  $x_i$  by 1. The question is: Is the resulting placement an optimal assignment for the modified problem? Unfortunately, the answer is “not necessarily.” Consider the case in which we have two servers, each of capacity 2 VMs, and three clients each having a single VM. A possible  $\pi$  in this case is for the VMs of clients 1 and 2 to co-reside on server 1 and for the VM of client 3 to be on server 2. If we now remove the VM of client 3 from the problem and decrement the capacity of server 2 by 1, we observe that the resulting placement is not optimal. The optimal placement places client 1's VM and client 2's VM on different servers.

We then asked if some special sub-cases of the problem exhibit optimal substructure. For example, consider the case in which the total number of VMs across clients is equal to the total capacity across servers, i.e.,  $\sum_i x_i = \sum_j s_j$ . In such a sub-case, a placement fills all servers to capacity. This sub-case does not exhibit optimal substructure either. A counterexample for the  $\langle NR, NC \rangle$  case is shown in Figure 3.3.



**Figure 3.3.** An example to show that a sub-case of the problem lacks optimal substructure. Consider the top figure in the  $\langle NR, NC \rangle$  model. Each of the three servers has a capacity of 3 VMs. The placement shown is optimal resulting in the least total information leakage. If we remove one of the VMs of client 2 from server 3, and decrement the capacity of server 3 by 1 VM we end up with the sub-optimal placement shown in the middle figure. The optimal placement is shown in the bottom figure.

**Open Problems** As we suggest in the immediately preceding discussions, while we have resolved some questions, we leave a number of others open. We list some of these here.

1. Is the Closed Migration problem tractable under models other than  $\langle R, C \rangle$ ? We know that it lies in **NP**, but does it lie in **P**? We emphasize (see Theorem 3.3.3 in Section 3.3.2) that the general problem, comprising both open and closed systems, is indeed NP-hard under all four models.
2. Is the problem we consider strongly **NP**-hard, or only weakly? Note that in Section 3.3.2, the problem from which we carry out a reduction, Set Partition, is weakly **NP**-hard only.
3. Is an instance of the optimization problem efficiently approximable? A corresponding question for cloud-scheduling that targets utilization, classical Bin-Packing, for example, is known to be efficiently approximable within a constant factor [26].

## 3.4 Analysis of Prior Approaches

In this section we revisit and analyze the work of Moon et al. [5]. To our knowledge, that work is the first to propose the models of information leakage that we adopt. That work considers three approaches: a mapping to ILP, an initial design for a greedy algorithm called Baseline Greedy, and a final greedy algorithm called Nomad. We describe and address each, in turn, in Sections 3.4.1–3.4.3 below.

### 3.4.1 Mapping to ILP

The mapping to ILP of Moon et al. [5] comprises eight sets of constraints plus a constraint that expresses the optimization objective. The objective for a solver is to decide the values of several binary indicator variables that are used to encode placement. That is, there is a variable which takes value 1 if and only if a particular VM of a client resides on a particular server in the optimal placement. An example of a constraint on those variables is that the sum of 1 values for those variables for a server, across all clients, should not exceed the capacity of the server. Similarly the sum of 1 values for those variables for a client, across all servers, should equal the number of VMs the client has. The solver is asked to minimize an expression for information leakage that uses the binary indicator variables.

In assessing the mapping to ILP, we distinguish two properties: correctness and efficiency. We deem such a mapping to be correct if it is a reduction. That is, with the mindset that we map a decision version of the problem to a decision version of ILP, does the mapping have the property that the input instance is true if and only if the output instance is true? The mapping of Moon et al. [5] does appear to be a reduction. We deem the mapping to be efficient if there exists a polynomial-time algorithm that computes it. Unfortunately, the mapping is not.

**Theorem 3.4.1.** *In the worst-case, Moon et al. [5]’s mapping to ILP results in an output whose size is exponential in the size of the input, and an algorithm to compute the mapping runs in exponential-time.*

*Proof.* We revisit our discussion on encoding placement in Section 3.3.3. As we discuss there, under the  $\langle R, C \rangle$  model, it suffices to record the number of VMs of each client on a server, rather than their identities. Under such an encoding, the size of an input placement is  $\Theta(n.k.\log \max_i \{x_i\})$  bits, where  $n$  is the number of clients,  $k$  is the number of servers and  $x_i$  is the number of VMs the  $i^{\text{th}}$  client has. Moon et al.’s ILP approach involves constraints over binary indicator variables that are used to encode a placement. For every VM and every server, their approach uses a binary indicator variable that indicates if that particular VM is placed on that particular server. Such a placement is of size  $\Theta(n.k.\max_i \{x_i\})$  bits. A case that elicits the worst case is when each of  $n$  and  $k$  are constant in  $x_i$ . Such cases can certainly arise in practice as the number of clients  $n$  and the number of servers  $k$  can be independent from the number of VMs that a client has. If we use  $c$  to represent a constant and  $m = \max_i x_i$ , then in the worst-case, Moon’s reduction results in an output size of  $\Theta(cm)$ , while the input placement under the compact encoding has a size in the worst-case of  $\Theta(c \log m)$ . The former is exponential in the latter, specifically,  $m = 2^{\log m}$ . Hence in the worst-case the choice of using binary indicator variables results in an exponential explosion in the size of the input.  $\square$

Our assessment of their mapping to ILP is motivated by the observation there that a commercial ILP solver takes longer than a day for even modestly sized inputs. Our observations above, and empirical results from our reduction to ILP (see Section 3.5 and [1]) suggest that this inefficiency in the mapping may be the cause.

### 3.4.2 Initial Design: Baseline Greedy

Moon et al. [5] use the observation that their mapping to ILP performs poorly in practice as motivation for the design of a greedy approach. In Baseline Greedy, in addition to all the inputs specified in the general problem, there is an additional input that enumerates the types of migration-moves allowed. A free-insert is when a single VM is moved from one server to another. A  $k$ -way swap of VMs  $\langle v_a, v_b, v_c, \dots, v_k \rangle$  means that VM  $v_a$  takes the place of  $v_k$ ,  $v_b$  takes the place of  $v_a$ ,  $v_c$  takes the place of  $v_b$ , and so on. A  $k$ -way swap incurs  $k$  migrations. Based on the types of moves allowed and the migration budget, Baseline Greedy generates a

list of possible moves and computes the reduction in leakage that results from each. The move that yields the greatest decrease in leakage is chosen and the current placement is changed to reflect this move. This happens iteratively until the migration budget runs out.

**Theorem 3.4.2.** *Baseline Greedy runs in time  $\Theta(m!)$  in the worst-case, where  $m = \sum_{i \in [1, n]} x_i$ . In the worst-case this is strictly worse than a brute force algorithm that tries every placement to determine which is optimal.*

*Proof.* We first observe that given  $i$  VMs, there are  $i!$  possible  $i$ -way swaps between them. The worst-case run-time for Baseline Greedy is when the set of allowed moves includes all possible  $i$ -way swaps, where  $i = 1, 2, \dots, m$ , and  $m$  is the total number of VMs in the instance. Hence the worst-case run-time of Baseline Greedy is  $\Theta(m!)$ . Next we observe that there are  $2^{mk}$  possible placements because for each of the  $m$  VMs and each of the  $k$  servers, the VM is either placed on that server or not. A brute force algorithm that tries all possible placements therefore has a run-time of  $\Theta(2^{mk})$ . If  $m \leq k$ , then an optimal placement that incurs zero leakage is one that places each VM on a separate server. If  $m > k$ , then  $m$  cannot be a constant if  $k$  is unbounded, but  $k$  can be a constant when  $m$  is unbounded. Hence in the worst-case,  $k$  is a constant, and the brute-force algorithm takes time  $\Theta(c^m)$ , where  $c = 2^k$ . For any constant  $c$ ,  $c^m = o(m!)$ , that is,  $\Theta(m!)$  is strictly worse than  $\Theta(c^m)$ .  $\square$

We acknowledge that Baseline Greedy is used only as a starting point, and its inefficiency is recognized by Moon et al. [5], albeit not as precisely as we do. An identification of an upper-bound for computational complexity helps with the identification of a more appropriate starting point for the design.

### 3.4.3 Final Design: Nomad

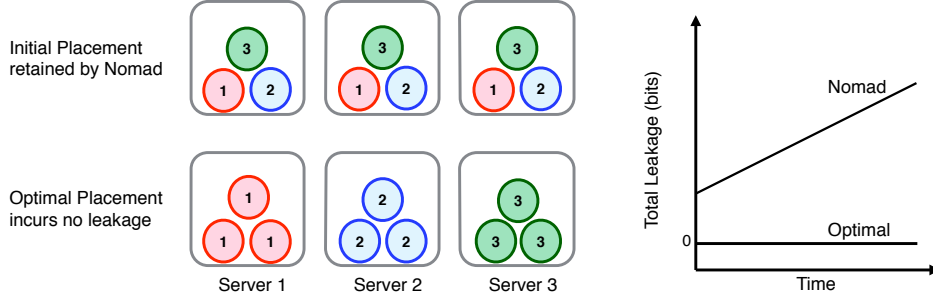
Moon et al.’s final design is a greedy algorithm called Nomad [5]. Nomad is Baseline Greedy with the following modifications. (1) Instead of recomputing the benefit for each move after a move is selected, the benefit is only recomputed for the moves involving client pairs which were affected by the selected move. For the leakage models that involve a maximization (i.e.,

“max”) term in the computation of information leakage, this is made possible by approximating the “max” operation with an exponential function. (2) Servers are grouped into clusters. Each client is assigned to a cluster and can only migrate within it. A move therefore only affects clients whose VMs reside in the same cluster in which the move was made. (3) The types of allowed moves are restricted to free-inserts and 2-way swaps only. (4) The entire move table is populated at the beginning of an epoch. Then, the move set is traversed starting from the move that gives the most benefit. If the claimed benefit from the time that benefit was computed lies within 95% of the current benefit and a move is feasible, then that move is made. If not, the move is re-inserted with an updated benefit.

In summary, at the beginning of each epoch, Nomad does the following:

1. It enumerates all possible free-inserts and 2-way swaps in a move table  $M$  and for each move, it computes its benefit as the further decrease in information leakage the move provides as compared to retaining the placement from the previous epoch.
2. It chooses the move that provides the greatest benefit, updates the current placement with this move, and decreases the input migration budget accordingly, i.e., by 1 if the chosen move was a free-insert and by 2 if the chosen move was a 2-way swap.
3. Then it iteratively does the following until the migration budget for the epoch runs out:
  - It finds the move  $z$  in  $M$  that gives the next greatest benefit.
  - It computes the benefit provided by  $z$  under the updated placement.
  - If the newly computed benefit is within 95% of the old benefit recorded in  $M$  for  $z$ , it updates the current placement with  $z$  and decreases the migration budget accordingly. Otherwise, it inserts the newly computed benefit for  $z$  in  $M$ .

In Nomad, as the moves are restricted to free-inserts and 2-way swaps, the running time is  $\Theta(n^2)$  in the worst-case. Given that the problem is **NP**-hard (see Section 3.3.2), and the premise that  $\mathbf{P} \neq \mathbf{NP}$ , no polynomial-time algorithm exists which can give us the optimal value for every input. An important question we then ask is: what is the trade-off that is



**Figure 3.4.** An example of an instance for which Nomad performs no migrations, even though a placement incurring no leakage is possible. There are 3 servers, each having a capacity of at least 3 VMs, and 3 clients, each having 3 VMs. Under the  $\langle NR, NC \rangle$ ,  $\langle R, NC \rangle$  and  $\langle NR, C \rangle$  models, no free-insert or pair-wise swap gives us a reduction in leakage, hence Nomad makes no moves. The optimal placement is shown at the bottom of the figure and incurs no leakage.

incurred by Nomad in exchange for its efficiency? We answer this question in Theorem 3.4.3 below. In the statement of the theorem, we refer to a *class* of inputs. We define it as follows. We observe that the number of servers, call it,  $k$ , is an input to the problem. Each value of  $k \geq 2$ , corresponds to a class of inputs. We point out that within each such class of inputs, there exist infinitely many inputs. This is because the capacity of each server is unbounded in any input.

**Theorem 3.4.3.**  *$P \neq NP$  implies that there exist infinitely many classes of inputs, each with infinitely many inputs, for which a reduction in information leakage is possible, but for which Nomad provides no decrease in information leakage whatsoever, even when unconstrained by a migration budget.*

We emphasize that the above theorem says that Nomad does not even provide a decrease in information leakage, let alone minimize it. We emphasize also, that the theorem is true for all four leakage models.

We prove Theorem 3.4.3 in four stages, which are Lemmas 3.4.4, 3.4.5, 3.4.6, and 3.4.7 below. In Lemma 3.4.4, we introduce a new, related problem, that we call the “Reduce Information Leakage for Two Servers” problem. We show that it is intractable, i.e., **NP**-hard under polynomial-time Turing reductions. With this result as the basis, we show, progressively, that there exists at least one input with two servers only for which a reduction in information-



leakage is possible, but Nomad provides none (Lemma 3.4.5), that there then must exist infinitely many two-server inputs each for which a reduction in information leakage is possible, but Nomad provides none (Lemma 3.4.6), and, that for every two-server input, there exists an  $n$ -server input for every  $n \geq 2$ , for which Nomad provides no reduction in information-leakage (Lemma 3.4.7). We then tie everything together for the proof of Theorem 3.4.3.

We begin by defining the Reduce Information Leakage for Two Servers problem. Given a current placement of VMs on two servers, output a different placement that results in strictly less information leakage than the current placement if one exists, otherwise output the current placement. In the following lemma, we establish that this Reduce Information Leakage for Two Servers problem is **NP**-hard under polynomial-time Turing reductions [27].

A polynomial-time Turing reduction from a problem  $A$  to a problem  $B$  is an algorithm that solves problem  $A$  using a polynomial number of calls to a subroutine, or oracle, for problem  $B$ , and polynomial time outside of those subroutine calls [27]. Such a reduction is thought to be weaker than the polynomial-time many-to-one reduction we customarily use to show that a decision problem is **NP**-hard. However, the fact that a problem is **NP**-hard under polynomial-time Turing reductions is also widely accepted as evidence that a problem is intractable [27]. We point out also that given the manner in which we have posed the Reduce Information Leakage for Two Servers problem, it is not a decision problem, i.e., its output is not a bit that corresponds to ‘true’ or ‘false.’ Therefore, the use of a polynomial-time Turing reduction seems appropriate to show intractability.

**Lemma 3.4.4.** *The Reduce Information Leakage for Two Servers problem is **NP**-hard under polynomial-time Turing reductions.*

*Proof.* We reduce from the optimization version of the Initial Placement problem (see Section 3.3.2). Recall that in Initial Placement, we are given a list of client VMs,  $\{x_1, x_2, \dots, x_n\}$  and two servers of equal capacity such that the total capacities of the two servers is equal to the total number of VMs, i.e.  $s_1 = s_2 = \sum_i x_i/2$ . We are asked to find a placement of VMs across the two servers that minimizes information leakage. Given a subroutine for the Reduce Information Leakage for Two Servers problem, we devise the following algorithm for Initial Placement.

We take the Initial Placement instance, and place  $s_1$  VMs on Server 1 and place the remaining on Server 2 using some (any) efficient algorithm. This gives us a current placement for the Reduce Information Leakage for Two Servers instance. We invoke the subroutine for the Reduce Information Leakage for Two Servers problem with this instance as input. If it outputs a different placement than the current placement, we use this output from the subroutine as the current placement for the next query to the subroutine. We stop when the output placement is identical to the current placement. This placement is the optimal placement for the Initial Placement instance. As there are potentially multiple possible placements that can reduce information leakage, a question that arises is: how can we be certain that the intermediate placements output by the subroutine will eventually converge to an optimal placement?

In an answer to this question, we observe that for the case of two full servers, for reducing information leakage, the current placement is immaterial in that there always exists a sequence of 2-way swaps from any current placement to an optimal placement. That is, for every VM on Server 1 in the current placement that has to be on Server 2 in an optimal placement, there exists a VM on Server 2 in the current placement that has to be on Server 1 in an optimal placement.

Another question that arises is, how can we be certain that the reduction calls the Reduce Information Leakage subroutine a polynomial number of times only? This follows because the information leakage that results from the placement that is used in the first call to the subroutine is polynomial in the size of the input. The output of each call to the oracle will in the very least, reduce the leakage by 1. Hence the total number of calls to the oracle will be polynomial in the size of the input.

Thus, the Reduce Information Leakage for Two Servers problem is **NP**-hard under polynomial-time Turing reductions. □

**Lemma 3.4.5.** *There exists an input with only two servers, for which Nomad makes no moves, but a reduction in information leakage is possible.*

*Proof.* Our proof is by contradiction. We first list the facts we know, or have established, and the assumption we make for the purpose of contradiction.

1.  $\mathbf{P} \neq \mathbf{NP}$ ,
2. The Reduce Information Leakage for Two Servers problem is **NP**-hard (Lemma 3.4.4),
3. Nomad is a polynomial-time algorithm, even when unconstrained by a migration budget,
4. Nomad can make free-inserts and two-way swaps,
5. If Nomad makes a move on some input placement, the resulting placement incurs strictly less information leakage than the input placement, and,
6. If Nomad makes no moves on some input placement, then that input placement incurs the minimum possible information leakage.

Item (1) is the customary assumption that the containment of  $\mathbf{P}$  in  $\mathbf{NP}$  is strict, Items (2)–(5) are facts that we know or have established, and Item (6) we assume for the purpose of contradiction.

The proof now is as follows. Items (4)–(6) imply that Nomad is an algorithm for the Reduce Information Leakage for Two Servers problem. But this, together with Item (3) implies that there exists a polynomial-time algorithm for the Reduce Information Leakage for Two Servers problem. This contradicts the conjunction of Items (1) and (2). Thus, Item (6) is false. That is, there exists at least one input for which a reduction in information leakage is possible, but Nomad makes no moves and therefore provides no reduction in information leakage.  $\square$

**Lemma 3.4.6.** *There exist infinitely many inputs, each with only two servers, for which a reduction in information leakage is possible, but Nomad makes no moves.*

*Proof.* We establish this lemma by contradiction as well. In Lemma 3.4.5, we establish that there exists such an input. Assume that only finitely many such inputs exist. Pick the largest-sized such input, and let this input's size be  $n_0$ . This implies that for all inputs of size  $n > n_0$ , Nomad satisfies Item (6) from the proof for Lemma 3.4.5. But this in turn implies that there exists an improved version of Nomad, call it NomadPrime, which works as follows. It first checks whether the size of the input is  $\leq n_0$ . If yes, it invokes an algorithm to correctly identify a placement that minimizes information leakage. This algorithm, for inputs

of size  $\leq n_0$ , is not necessarily efficient; e.g., it may, by brute-force, simply try every possible placement. If the input size  $> n_0$ , it invokes Nomad.

Then, NomadPrime is a polynomial-time algorithm, because  $n_0$  is a constant, and for all  $n > n_0$ , it runs in time polynomial in the size of the input. And NomadPrime is then a polynomial-time algorithm for Reduce Information Leakage for Two Servers. This contradicts the conjunction of Items (1) and (2) in the proof of Theorem 3.4.5.  $\square$

**Lemma 3.4.7.** *There exists a one-to-one mapping,  $\psi$ , between the set of inputs with only two servers for which a reduction in information leakage is possible, and the set of inputs with some  $k \geq 2$  servers for which a reduction in information leakage is possible, with the following property. If Nomad makes no moves on the two-server input,  $i$ , then Nomad makes no moves on the  $k$ -server input  $\psi(i)$ .*

*Proof.* By construction — we present such a mapping,  $\psi$ . If  $k = 2$ , we simply pick the identity mapping as  $\psi$ . Otherwise,  $k > 3$ . We are given as input  $i$ , which comprises two servers. We map that to,  $\psi(i)$ , an input with  $k$  servers. Two servers in  $\psi(i)$  are identical to the two servers in the input  $i$ . For each of the remaining  $k - 2$  servers, we pick some capacity. And we introduce a new client, call it  $y$ . By “new client,” we mean that there are no VMs of the client  $y$  in the two servers in the two-server input  $i$ . We fill each of the  $k - 2$  servers with VMs of that new client,  $y$ . We now claim that if Nomad makes no moves on input  $i$ , then it makes no moves on input  $\psi(i)$ .

We prove this as follows. Given the input  $\psi(i)$  constructed as above, we know that a decrease in leakage is possible for it - a candidate placement is to make no changes to the new servers, and for the original two servers, use the placement that decreases leakage. For  $\psi(i)$ , we show that under each of the four leakage models, Nomad will not decrease leakage as there is no free-insert or 2-way swap that results in a decrease in leakage.

First we observe that under all four models, the information leakage arising from the newly added servers is zero, since all of them contain VMs belonging to the same client,  $y$ . Therefore Nomad will not make any 2-way swaps between VMs of these new servers as this would result in no change in information leakage. Nomad will also not make any free-inserts or

2-way swaps between VMs of the two original servers as otherwise it would contradict the fact that the placement on the original two servers is one for which a decrease in leakage is possible, but for which Nomad makes no moves. Nomad will also not make any free-insert of a VM from a newly added server to one of the two original servers, as this would increase co-residency and hence information leakage. The only other kind of move to consider is a 2-way swap between a VM on a new server and a VM on one of the two original servers. Let the new and old server involved in the swap be denoted by  $\sigma_{new}$  and  $\sigma_{old}$  respectively. Let  $\sigma_{new}$  be filled with VMs of the new client  $y$ . Let one such VM be denoted by  $VM_y$ . Let  $x$  be one of the clients having at least one VM on  $\sigma_{old}$ . Let one such VM be denoted by  $VM_x$ . Now assume Nomad swaps the placement of  $VM_x$  and  $VM_y$ .

We will show that under all four models, the leakage incurred by the new placement (after the swap) is greater than or equal to the leakage incurred by the original placement, and hence Nomad will not make the move. It suffices for us to show that this is the case when the capacity of  $\sigma_{new}$  is 1. Because if this is true, then even if the capacity of  $\sigma_{new}$  is more than 1, the new co-residency between clients  $x$  and  $y$  on  $\sigma_{new}$  in the new placement can only further increase information leakage from the case in which the capacity of  $\sigma_{new}$  is 1.

We show that for all four leakage models, the maximum decrease in leakage from removing  $VM_x$  from  $\sigma_{old}$  is less than or equal to the minimum increase in leakage incurred from placing  $VM_y$  on  $\sigma_{old}$ .

Consider the  $\langle NR, NC \rangle$  model. If in the original placement the number of clients other than client  $x$  on  $\sigma_{old}$  is  $w$ , then the maximum decrease in leakage that results from moving  $VM_x$  out of  $\sigma_{old}$  is when these  $w$  clients are not co-resident with any other VM of  $x$  besides  $VM_x$ , and it is  $2w$  bits;  $w$  bits from client  $x$  and  $w$  bits to client  $x$ . The minimum increase in leakage that results from placing  $VM_y$  on  $\sigma_{old}$  is  $2w$  bits because now client  $y$  is co-resident with at least  $w$  other clients. It is more than  $2w$  bits if client  $x$  has VMs other than  $VM_x$  on  $\sigma_{old}$  in the original placement, because then client  $y$  is co-resident with client  $x$  also. Hence Nomad will not make the swap under the  $\langle NR, NC \rangle$  model.

Consider the  $\langle R, C \rangle$  model. If in the original placement the number of VMs on  $\sigma_{old}$  belonging to clients other than client  $x$  is  $z$ , then the decrease in leakage that results from moving  $VM_x$

out of  $\sigma_{old}$  is  $2z$  bits;  $z$  bits from client  $x$ , and  $z$  bits to client  $x$ . The minimum increase in leakage that results from placing  $VM_y$  on  $\sigma_{old}$  is  $2z$  bit;  $z$  bits from client  $y$  and  $z$  bits to client  $y$ . It is more than  $2z$  bits if client  $x$  has VMs other than  $VM_x$  on  $\sigma_{old}$  in the original placement. Hence Nomad will not make the swap under the  $\langle R, C \rangle$  model.

Consider the  $\langle NR, C \rangle$  model. If in the original placement the number of clients other than client  $x$  on  $\sigma_{old}$  is  $w$ , and the number of VMs on  $\sigma_{old}$  belonging to clients other than client  $x$  is  $z$ , then the maximum decrease in leakage that results from moving  $VM_x$  out of  $\sigma_{old}$  is when these  $w$  clients are not co-resident with any other VM of  $x$  besides  $VM_x$  and it is  $w + z$  bits;  $w$  bits to client  $x$  and  $z$  bits from client  $x$ . The minimum increase in leakage that results from placing  $VM_y$  on  $\sigma_{old}$  is  $w + z$  bits;  $w$  bits to client  $y$  and  $z$  bits from client  $y$ . It is more than  $w + z$  bits if client  $x$  has VMs other than  $VM_x$  on  $\sigma_{old}$  in the original placement. Hence Nomad will not make the swap under the  $\langle NR, C \rangle$  model.

As the  $\langle R, NC \rangle$  model is symmetric to the  $\langle NR, C \rangle$  model, the result for it follows from the result for the  $\langle NR, C \rangle$  model.

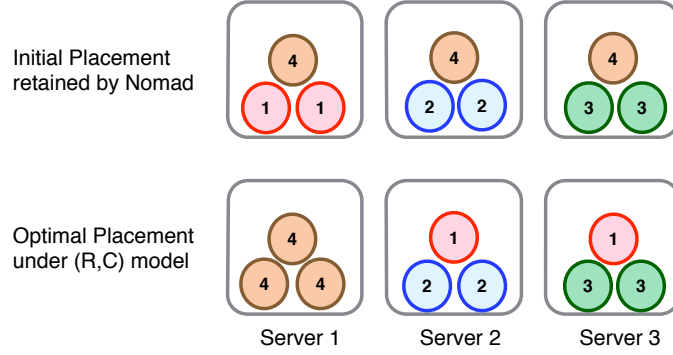
□

**Proof. (for Theorem 3.4.3)**

Lemma 3.4.6 establishes that there exist infinitely many inputs in the class of inputs,  $k = 2$ , for which a reduction in information leakage is possible, but Nomad provides none. By Lemma 3.4.7, each such input can be mapped uniquely to an input in every other class of inputs. The proof follows from the fact that the number of classes of inputs, which corresponds to the number of servers,  $k$ , is unbounded, in any particular input.

□

**Examples** As a concrete example of a class of inputs for which Nomad provides no benefit, consider the following for the  $\langle NR, NC \rangle$ ,  $\langle R, NC \rangle$  and  $\langle NR, C \rangle$  models. We have  $n$  servers, each of capacity of at least  $n$ , and  $n$  clients each with  $n$  VMs, where  $n \geq 3$ . In the initial placement, each server has a VM that belongs to each client. Figure 3.4 shows the current placement when  $n = 3$ . The optimal leakage is 0, obtained from the placement in which all the VMs of client  $i$  are on server  $i$ . However on running Nomad on this instance, with an



**Figure 3.5.** An example to show the sub-optimality of Nomad for the  $\langle R, C \rangle$  model. The capacity of each server is at least 3 VMs. Clients 1, 2 and 3 have two VMs each, and Client 4 has three VMs. The placement at the top of the figure is the current placement. There is no free-insert or 2-way swap that results in a reduction in leakage. Hence Nomad makes no moves even when unconstrained by a migration budget. The optimal placement is shown in the bottom of the figure.

unbounded migration budget, we find that no free-insert or 2-way swap yields a reduction in leakage. So, Nomad does not change the current placement, which is sub-optimal. Indeed, for the  $\langle NR, NC \rangle$  model, the current placement incurs the worst possible information leakage.

As another example, for the  $\langle R, C \rangle$  model, consider the following class of inputs. There are  $n$  servers, each with capacity of at least  $n$ , and  $n + 1$  clients. Clients  $1, 2, \dots, n$  have  $n - 1$  VMs each, and client  $n + 1$  has  $n$  VMs. In the current placement, server  $i$  has the VMs:  $\langle i, 1 \rangle, \langle i, 2 \rangle, \dots, \langle i, n - 1 \rangle, \langle n + 1, i \rangle$ , where  $\langle x, y \rangle$  corresponds to VM  $y$  of client  $x$ . There is no free-insert or 2-way swap which yields a reduction in leakage for this placement. Hence Nomad makes no changes to the current placement. The current placement and optimal placement for  $n = 3$  are shown in Figure 3.5. We have tried this input against the Nomad software which we have been provided, and confirmed the above observation.

### 3.5 Reductions to ILP and CNF-SAT

Given that Nomad provides no decrease in information leakage for infinitely many inputs, we revert to the two approaches that we discuss in this section. One is an efficient reduction

to ILP and the other is an efficient reduction to CNF-SAT. Our primary intent is to identify whether we can indeed realize software that is efficient in practice, and provides the security guarantee of minimizing information leakage, notwithstanding the **NP**-hard worst-case for computational complexity. Both approaches are efficient algorithms given access to oracles for ILP and CNF-SAT, respectively. We chose to invest in both approaches because there can be differences in the performance of individual solvers for different classes of problem instances. Similar to prior work, we also focus on the  $\langle R, C \rangle$  leakage model because it is the most conservative of the four models — replication and collusion are both advantageous to the attacker.

**Input:** An instance of the scheduling problem,  $\langle X, S, A, m, F \rangle$

**Output:** An ILP instance: (i) a set of linear constraints  $C$ , and, (ii) an objective function  $Ob$

```

1  $C \leftarrow \sum_{i \in [1, n]} \sum_{j \in [1, k]} |f_t(i, j) - f_{t-1}(i, j)| \leq 2 \times m$ 
2 foreach  $s_j \in S, j \leftarrow 1$  to  $k$  do
3    $C \leftarrow C \cup \sum_{i \in [1, n]} |f_t(i, j)| \leq s_j$ 
4 foreach  $i \leftarrow 1$  to  $n$  do
5    $C \leftarrow C \cup \sum_{j \in [1, k]} |f_t(i, j)| = x_i$ 
6  $Ob \leftarrow \min \sum_{\substack{(c, c') \\ c, c' \in [1, n] \\ c \neq c'}} \sum_{t' \in (t-\Delta, t]} \sum_{j \in [1, k]} f_{t'}(c, j) \times f_{t'}(c', j)$ 
7 return  $\langle C, Ob \rangle$ 

```

**Algorithm 1:** Reduction to ILP for the  $\langle R, C \rangle$  model for information leakage.  $C, Ob$  are the outputs,  $|\cdot|$  is absolute value, and the other symbols are explained in Table 3.2 in Section 3.2. Line (1) ensures that the migration budget,  $m$ , is respected. Lines (2)–(3) ensure that the capacity of a server,  $s_i$ , is not exceeded. Lines (4)–(5) ensure that every VM of each client is placed on exactly one server. The objective function on Line (6) minimizes the total information leakage.



### 3.5.1 Reduction to ILP

Algorithm 1 is our reduction to ILP for the  $\langle R, C \rangle$  case. The other cases require minor modifications only, as we discuss in the following. The input to the reduction is an instance of the problem posed in Section 3.2, and the output is the corresponding ILP instance. The unknown in the constraints and the optimization objective, is a new placement — values for the  $n \times k$  variables  $f_t(i, j)$ , for each  $i \in [1, n], j \in [1, k]$ , for a given  $t$ , for the  $\langle R, C \rangle$  case. These values are the number of VMs of client  $i$  on server  $j$  in the upcoming epoch,  $t$ . For the other three models of information leakage, the unknowns are  $f_t(v, i, j)$ , where  $v \in [1, x_i]$ , which indicates whether VM  $v$  of client  $i$  is placed on server  $j$  in epoch  $t$ .

In Algorithm 1, Line (1) initializes the output set of constraints  $C$ , with a constraint that ensures that the migration budget,  $m$ , is not exceeded by whatever moves are needed to reach the new placement,  $f_t$ . If  $m$  is specified as  $\infty$  in the input, we initialize  $C \leftarrow \emptyset$  instead. Lines (2)–(3) add constraints to ensure that the capacity of each server,  $s_j$ , is not exceeded in the new placement. Lines (4)–(5) add constraints to ensure that every VM of each client is placed on exactly one server. The objective function,  $Ob$ , that is instantiated in Line (6), ensures that we minimize the total information leakage with the new placement. The only nuance here is that the objective function,  $Ob$ , for the  $\langle R, C \rangle$  case, is not a set of linear constraints, but rather quadratic constraints. However, Quadratic Constraint Programming is known to be in **NP** as well [28] — we have designed and implemented a reduction from quadratic to linear constraints via bit-wise multiplication. Our reduction is based on the standard algorithm used for multiplication in which one multiplies the multiplicand by each digit of the multiplier and then adds up all the properly shifted results. This algorithm and hence our reduction runs in quadratic time.

Algorithm 1 runs in polynomial-time; its running-time is:  $O\left(\binom{n}{2} \cdot \Delta \cdot k \cdot \log_2^2 \max_i x_i\right)$ , where  $n$  is the number of clients,  $\Delta$  is the number of epochs in the sliding-window,  $k$  is the number of servers, and  $\log_2 \max_i x_i$  is the maximum number of bits we need to express the number of VMs per client. The reason is that the running time is dominated by Line (6) — the above expression for the running-time comes directly from the three summations and the product. This bound is not tight; more efficient reductions can exist from, for example, more compact

encodings for placement.

### 3.5.2 Reduction to CNF-SAT

We reduce the decision version of the scheduling problem to CNF-SAT. That is, in addition to the inputs  $X, S, A, m$  and  $F$ , we take as input an integer,  $q$ . This input  $q$  represents a threshold for the information leakage we want to meet. That is, we ask if there exists a new placement whose information leakage is at most  $q$ . To obtain a solution to the optimization version, we perform a binary-search on  $q$ . In other words, if adopting the current placement for the next epoch results in a total information leakage of at most  $q$ , we set the threshold to  $q/2$ , perform a reduction to CNF-SAT and query the solver again. If the solver returns with ‘yes’, then we set the threshold to  $q/4$  and so on. Our reduction of the problem considered in this thesis to CNF-SAT is via CIRCUIT-SAT, the satisfiability problem for boolean circuits [29]. Once we reduce to CIRCUIT-SAT, we adopt a well-known linear-time reduction from CIRCUIT-SAT to CNF-SAT [29]. Algorithm 2 is our reduction to CIRCUIT-SAT.

Our reduction is based on the work of Sinz [30] and Mousavi [31]. Those pieces of work articulate how constraints of the form we need to address can be encoded as boolean circuits. We refer the reader to those pieces of work for more comprehensive discussions, and focus on our scheduling problem here.

The reduction to CIRCUIT-SAT, as Algorithm 2 indicates, is similar to the reduction to ILP. In lieu of constraints that we express in ILP, we express constraints using boolean circuits, whose basic elements are AND, OR and NOT gates. The output of the reduction in Algorithm 2 is a boolean circuit with input wires for a solver to set in a manner that the single output wire evaluates to 1, i.e., the circuit is satisfied. As in the reduction to ILP in Section 3.5, the unknown which the solver determines is a new placement. Thus, the input wires correspond to, or encode, a placement. The input wires,  $w_{i,j}$ , as Lines (1)–(3) in the algorithm indicate, represent a placement that satisfies all the constraints encoded by the circuit. The manner in which they represent a placement is that a placement for us is a set of  $n \times k$  integers,  $f_t(i, j)$ , for each  $i \in [1, n], j \in [1, k]$ , given  $n$  clients, and  $k$  servers. That is,  $f_t(i, j)$  is the number of

**Input:** An instance of the decision version of the scheduling problem

$\langle X, S, A, m, F, q \rangle$

**Output:** A CIRCUIT-SAT instance, i.e., a boolean circuit with a set of input wires and a single output wire

```

1  foreach  $i \leftarrow 1$  to  $n$  do
2      foreach  $j \leftarrow 1$  to  $k$  do
3          Form a set,  $w_{i,j}$ , of  $\log_2(x_i)$  wires
4  foreach  $s_j \in S$ ,  $j \leftarrow 1$  to  $k$  do
5      foreach  $i \leftarrow 1$  to  $n$  do
6          The  $w_{i,j}$  wires as input to an adder circuit,  $A_j$ 
7          A comparison circuit:  $A_j \leq s_j$ 
8  foreach  $i \leftarrow 1$  to  $n$  do
9      foreach  $j \leftarrow 1$  to  $k$  do
10         The  $w_{i,j}$  wires as input to an adder circuit,  $A_i$ 
11         A comparison circuit:  $A_i = x_i$ 
12 foreach  $i \leftarrow 1$  to  $n$  do
13     foreach  $j \leftarrow 1$  to  $k$  do
14         A circuit to determine  $d_{i,j} = |f_t(i, j) - f_{t-1}(i, j)|$ 
15     A circuit to ensure:  $\sum_i \sum_j d_{i,j} \leq 2 \cdot m$ 
16 A circuit that ensures:

```

$$\sum_{\substack{(c,c') \\ c,c' \in [1,n] \\ c \neq c'}} \sum_{t' \in (t-\Delta, t]} \sum_{j \in [1,k]} f_{t'}(c, j) \times f_{t'}(c', j) \leq q$$

```

17 Single-output AND of all the above

```

**Algorithm 2:** Reduction to CIRCUIT-SAT from a decision version of the scheduling problem, for the  $\langle R, C \rangle$  model. The inputs  $X, S, A, m, F$  are explained in Table 3.2 in Section 3.2. The input  $q$  is an information-leakage threshold.

VMs of client  $i$  that are to reside on server  $j$ .

This integer value,  $f_t(i, j)$  can be represented using at most  $\log_2(x_i)$  bits, where  $x_i$  is the total number of VMs of client  $i$ . This is the reason that in Line (3) of Algorithm 2, we have a  $\log_2(x_i)$ -sized set of wires corresponding to each  $i, j$ . Of course, some leading bits amongst this set of bits may be all 0, when VMs of a client are spread across several servers. But we need to account for the possibility that every VM of a client may end up resident on the same server.

Once we decide on these inputs wires, we can go ahead and express boolean-circuit constraints on them. The constraints are the same as for the reduction to ILP that we discuss in Section 3.5. Specifically, in Lines (4)–(7), we first, in the inner for loop in Lines (5)–(6), build adder circuits to add the number of VMs of each client that is resident on each server. Then, in the outer for loop of Lines (4) and (7), we use a comparison circuit to specify the constraint that the output of the adder circuits should be at most the capacity of each server,  $s_j$ . Note that the integer  $s_j$  is itself encoded as a circuit. It is easy to build a circuit of such a constant bit-string. E.g., a “1” can be encoded as “ $x \vee \neg x$ ”, and a “0” as “ $x \wedge \neg x$ .” While we have specified those using propositional logic, it is easy to translate them to circuits that use OR, AND and NOT gates in lieu of  $\vee$ ,  $\wedge$  and  $\neg$ , respectively.

Lines (8)–(11) similarly encode the constraint that each client resides on exactly one server, and Lines (12)–(15) ensure that the migration budget is not exceeded. Finally, Line (16) ensures that the threshold for information-leakage is met. In Line (17), we AND all the above circuits; the output of this AND is the output of the circuit. The question to a solver, now, is whether the input wires can be set to bit-values in a manner that the output wire evaluates to 1. This is possible if and only if all the constraints are satisfiable.

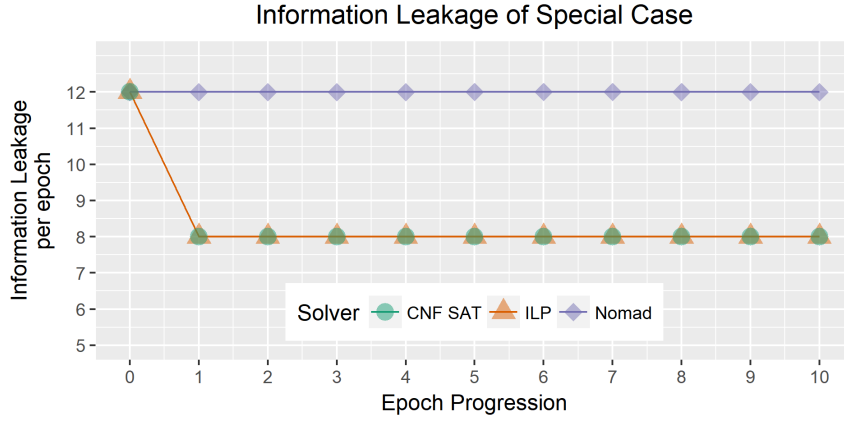
This is exactly the question we seek to answer. The running-time of the reduction to CIRCUIT-SAT is the same as the running-time of the reduction to ILP, i.e., polynomial in the size of the input. And, as we mention above, the reduction from CIRCUIT-SAT to CNF-SAT is linear-time.

### 3.5.3 Empirical Evaluation

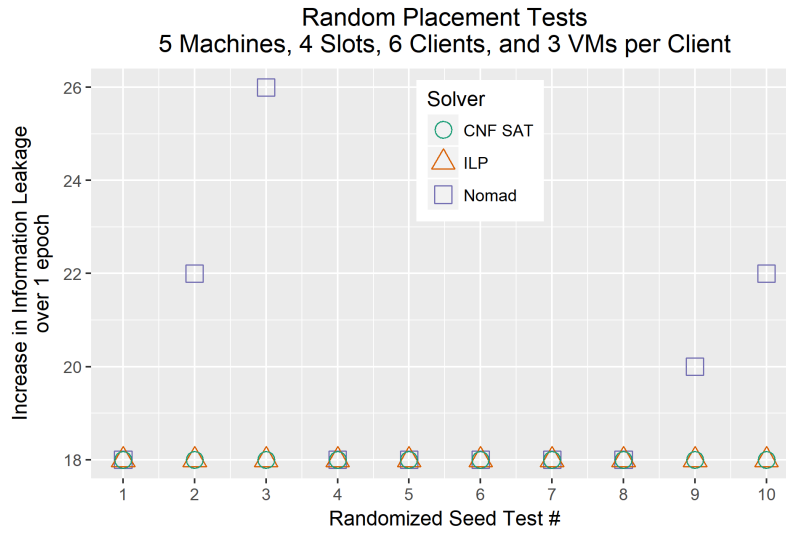
A limited empirical assessment of our two approaches and Nomad can be found in our paper [1], and in other work [8]. The intent of the empirical assessment was: (a) to validate empirically our analytical insights on the shortcomings of Nomad, and, (b) to understand the overhead of actually providing a security guarantee of minimizing leakage in practice. For the ILP approach, we used the CPLEX [32] solver as it was used in prior work [5]. For the CNF-SAT approach, we used Lingeling [33] as the solver. Below we summarize the results and takeaways from the assessment.

On (a), we found that our empirical observations concurred with our analytical insights on Nomad’s lack of security. Figure 3.5 in Section 3.4 shows an example of an input for which we expect Nomad not to make any moves even though a decrease in leakage is possible. Figure 3.6 shows the results from running this particular example in practice. We observe from the figure that for this input, the approaches based on ILP and CNF-SAT converged to the optimal leakage of 8 bits per epoch within a single epoch, but Nomad provided no decrease in information-leakage even when unconstrained by a migration budget. As another example, Figure 3.7 shows how each of the three approaches performed after 1 epoch when they were given random inputs that have a known non-zero optimal leakage. We observe from the figure that ILP and CNF-SAT consistently converged to the optimal information leakage, but Nomad’s performance was inconsistent — for 4 out of the 10 cases, it was unable to converge to the optimal leakage.

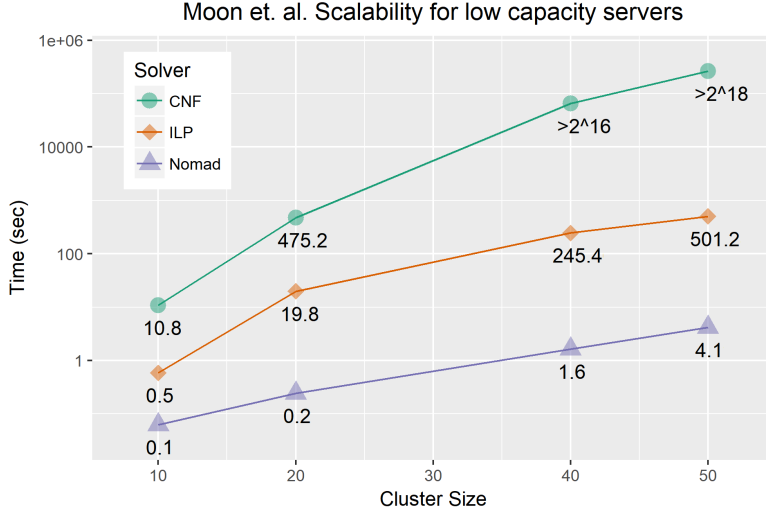
On (b), our results suggest that while both the ILP and SAT approaches converge to the optimal placement, the ILP approach is often faster. An ILP solver solves an optimization problem and therefore the ILP approach, given an instance of the scheduling optimization problem, comprises a single reduction to ILP along with a single call to an ILP solver. A SAT solver, on the other hand, solves a decision problem, and therefore the CNF-SAT approach comprises a binary search for the optimal leakage value. Each iteration of the binary search entails a reduction to CNF-SAT and a call to a SAT solver. The ILP approach’s faster performance than that of the CNF-SAT approach can be observed in Figure 3.8 which shows the results from a test that was conducted to assess the scalability of the three approaches.



**Figure 3.6.** This figure shows how information leakage varies when all three approaches are allowed to run 10 epochs on the placement map shown in Figure 3.5. The CNF-SAT and ILP solvers were both able to obtain the optimal placement in which a leakage of 8 bits is incurred per epoch. Nomad was unable to make any move. ©2020 IEEE.



**Figure 3.7.** Increase in information leakage in 1 epoch for 10 random initial placements for input instances of 5 servers with 4 VM slots each, and 6 clients with 3 VMs each. Each of the 10 tests were run over a duration of 1 epoch with an infinite migration budget. The optimal placement is the one resulting in leakage of 18 bits. ILP and CNF-SAT always converged to the optimal. Nomad's performance was inconsistent. ©2020 IEEE.



**Figure 3.8.** Scalability of Nomad, ILP, and CNF-SAT using the testing parameters presented in Moon et. al [5]. Cluster size of  $x$  means:  $x$  servers with capacity 4, and  $x$  clients with 2 VMs each. Total VMs in the cluster:  $\sum VM = 2 \cdot x$ . ©2020 IEEE.

Our approaches that provide security incur a higher overhead than Nomad (see Figure 3.8). However, this overhead, in practice, is not as high as suggested by prior work. We find that if an efficient reduction to ILP is devised and adopted, then the run-time is significantly lower than previously thought. For example, while prior work reports that their ILP approach takes longer than a day for cluster sizes of 40 and 50 [5], ours took only 4 and 8 minutes, respectively. This difference can be attributed to the result established in Theorem 3.4.1, i.e., the reduction to ILP in prior work can be exponential time in the size of the input for the  $\langle R, C \rangle$  model. In contrast, our reduction to ILP and CNF-SAT are polynomial-time. Whereas we are able to guarantee minimum information-leakage, it is unclear what security benefit more efficient approaches, particularly Nomad, actually provide.

The main takeaway from our empirical assessment is that security comes at a cost. This cost can be lowered by prudently leveraging tools such as constraint-solvers. But devising efficient reductions so they can be leveraged prudently can be a technical challenge. If we are to propose VM-migration as a viable technique to reduce information-leakage at scale, then

more research-investment is needed, e.g., into more efficient reductions, and solvers, before it can be deemed to be practical. Our results suggest that such a technique can be practical for relatively small clouds only — ones with 10's, but not 100's or 1000's, of servers. Approaches such as Nomad suggest that this technical challenge can be bypassed easily. But this is simply not true. The reasons for this lie at the very foundations of computing.

## 3.6 Related Work

We discuss work on cross-VM side-channels and defences against them.

**Cross-VM side-channel attacks** Ristenpart et al. [13] were among the first to show how an attacker can map the internal cloud infrastructure using network probing in order to achieve co-residency with a victim VM, and how the co-residency can be exploited to extract coarse-grained information such as keystroke patterns.

This was followed up by the work of Zhang et al. [16] who showed how a Prime+Probe side-channel attack on upper level cache can be used to extract more fine-grained information such as an ElGamal decryption key. In the prime stage, the attacker fills a portion of cache and then waits to allow the victim to access cache. In the probe stage, the attacker reloads the primed data and uses the probe time to decide whether the victim accessed that portion of cache. Prime+Probe attacks on last level caches that make use of huge page-sizes have also been studied [11, 9, 34]. More recent work shows a cross-core Prime+Probe attack on sliced non-inclusive caches [35].

Research has also shown how fine-grain information can be extracted using Flush+Reload techniques [17, 10, 15, 36]. In the flush stage, the attacker flushes out certain lines in memory and waits to allow the victim to access them. In the reload stage, the attacker accesses the flushed lines. If any such line does not take long to load, the attacker knows the line was accessed by the victim.

Other works have shown that sharing of same-content memory pages among co-resident clients



poses the threat of a memory disclosure attack [12, 14]. The attacker can identify which pages it shares with a victim and take advantage of differences in write times to gain information about the victim’s applications and OS.

Other work has shown side-channel attacks based on page faults against SGX-based shielded execution [37, 38]. In such an attack, a compromised OS allocates a minimum number of physical pages to the victim process, such that memory accesses spill out of the allocated set as much as possible, leaving a trace of page faults that leaks sensitive information.

Of course, other kinds of side-channels may exist. For example, the work of Xu et al. [37] considers the issue of excluding the operating system from the trusted computing base, and side-channels that can result as a consequence. As another example, the work of Shinde et al. [38] considers side-channels based on page-faults. Such attacks can certainly underlie cross-VM side-channel attacks. For example, if a benign VM runs a particular kind of application that allows a malicious VM that is co-resident to cause a page-fault, it may be vulnerable to the latter kind of attack.

**Defences** To counter such attacks, several defence strategies have been proposed. At a hypervisor level, Vattikonda et al. [18] proposed hiding a program’s execution time by eliminating fine grained timers. Kim et al. [39] used statistical multiplexing to protect against cache-based side-channel attacks in the cloud. They proposed a set of locked cache lines per core that are efficiently multiplexed amongst co-resident VMs and never evicted from the cache. Raj et al. [19] identified last level cache (LLC) sharing as one of the impediments to finer grain isolation, and proposed two resource management approaches to provide performance and security isolation - cache hierarchy aware core assignment and page colouring based cache partitioning. Shi et al. [40] proposed an approach that leverages dynamic cache colouring: when an application is doing security-sensitive operations, the VMM is notified to swap the associated data to a safe and isolated cache line. Varadarajan et al. [41] introduced a minimum run time (MRT) guarantee into the Xen scheduler, to limit the frequency of preemptions.

At a guest OS level, Zhang et al. [42] proposed a method by which a tenant can construct their

VMs to automatically inject additional noise into the timings that an attacker might observe from caches. Pattuk et al. [20] suggested partitioning cryptographic keys into random shares and storing each share in a different VM. Zhou et al. [43] presented a software approach which dynamically manages physical memory pages shared between security domains to disable sharing of LLC lines.

At a hardware level, some works have proposed modifying the cache architecture to obtain access randomization [44, 45] while others have explored the partitioning of cache [46]. In recent work, Liu et al. [47] demonstrated how to combat LLC side-channel attacks by using the Intel Cache Allocation Technology (CAT) to partition the LLC into a hybrid hardware-software managed cache.

As many side-channel attacks are based on the observances of memory access, a promising defence strategy is to use oblivious RAM (ORAM) to randomize data access patterns [48, 49]. When an adversary observes the physical storage locations accessed, the ORAM algorithm ensures that they have negligible probability of learning the true access pattern.

An approach that focuses on isolation is that of shielded execution, for example, by using Intel SGX enclaves [22]. Chen et al. [50] presented a software framework that enables a shielded execution to detect page-fault side-channel attacks.

Various scheduler-based defences have also been explored. Y. Zhang et al. [51] proposed periodically migrating VMs using methods based on game theory. Azar et al. [52] focused on mitigating co-location attacks, on the premise that they are a necessary first step to performing cross-VM attacks. Their approach assigns VMs to servers such that attack VMs are rarely co-located with target VMs. Their model is based on co-location rather than on information leakage. Han et al. [53] also studied how to minimize the attacker's possibility of co-locating their VMs with targets. They introduced a security game model to compare different VM allocation policies and advocated selecting an allocation policy statistically from a pool instead of having a fixed one. Bijon et al. [54] proposed an attribute-based constraints specification framework that enables clients to express essential properties of their resources as attributes. A constraints enforcement engine then schedules the VMs while respecting the conflicts specified by these attributes. Han et al. [55] proposed metrics which can be used to assess the security

of a VM allocation policy.

The work that is most closely related to ours is by Moon et al. [5]. They characterized information leakage models which can be used to pose the security-sensitive scheduling problem precisely and explore various approaches to address the problem. Our work differs from theirs in the following ways: (1) We analyze the problem's computational complexity, (2) Our reduction to ILP is polynomial time, while theirs can be exponential time for a certain encoding of input, (3) We also investigate the approach of reducing to CNF-SAT and employing a SAT solver, and (4) Our approaches guarantee security whereas their final approach, Nomad, does not.

## **3.7 Conclusions**

Our conclusions are discussed in Chapter 6, Section 6.1.

# Chapter 4

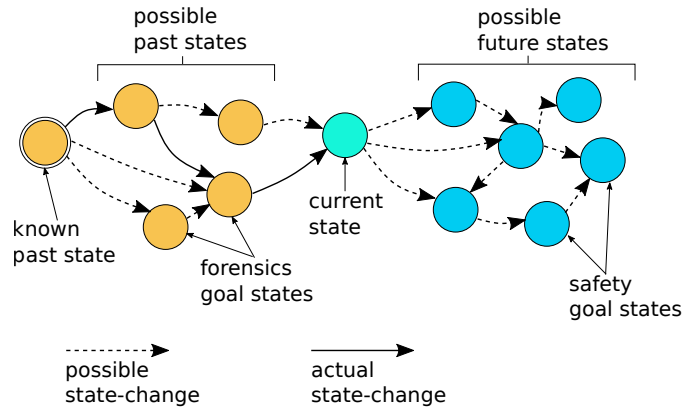
## Forensic Analysis for Access Control<sup>4</sup>

### 4.1 Introduction

In this work we unify two important topics in information security. *Forensic analysis* (or simply, *forensics*) seeks to answer questions about past states of a system. It is the process of identifying, preserving, analysing and presenting evidence in a manner that is legally acceptable [57]. In the context of computer and information security, it closely parallels the challenges faced by investigators at the scene of a possible crime. For example, given a chunk of bytes on a hard disk, we may ask whether those bytes are, or were, part of an image. We may ask also whether there was an attempt to delete that image from the disk. While there has been no prior work on forensic analysis specifically in the context of access control, there has been work on forensics in broader contexts such as intrusion detection and event reconstruction systems (see, for example, [58, 59, 60, 61, 62, 63] and Section 4.6).

---

<sup>4</sup>This chapter is based on work that is undergoing review [3]. My contributions are the following: posing the forensic analysis problem for access control (Section 4.1), formulating the problem (Section 4.2), the theorems and proofs demonstrating hardness results for HRU (Section 4.3.1) and ARBAC (Section 4.3.2), the algorithm for forensic analysis for Graham-Denning (Section 4.3.3), and the notion and theorems pertaining to goal-directed logging (Section 4.4). The case study in Section 4.5 was conducted by Huang Xiaowei and further details about it can be found in his MASc thesis [56].



**Figure 4.1.** Forensic vs. safety analysis. In safety analysis, we ask whether a future “goal” state with a certain property is reachable from the current state. In forensic analysis we ask if we could have reached, or did reach, the current state from a past “goal” state with a certain property, possibly delimiting our search with a known past state.

*Access control* deals with actions that a principal may exercise on a resource. It is recognized as an important aspect of the security of a system; indeed, its “centre of gravity” [64]. When a principal, such as a process of an operating system, seeks to exercise an action, such as read or write, on a resource, such as a digital file, the access control system ensures that the attempt is mediated. An *authorization policy* is used by the mediator to determine whether the attempt should be allowed. In realistic systems, this authorization policy may change over time. For example, Alice may now be disallowed from writing to a file that she used to be able to write to. As another example, a user or a file may be deleted from the system entirely.

**Forensics vs. safety** An authorization policy at any given moment can be perceived as an instance of a state in a state-change system. This is exactly the mindset that underlies *safety analysis*, a well-researched problem in access control (see Figure 4.1). As we discuss further in Section 4.6, there has been considerable work on safety analysis for access control (see, for example, [65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76]). The intent behind safety analysis is to determine whether an undesirable access may be authorized in some future state of the system by some sequence of actions, each of which changes the authorization policy.

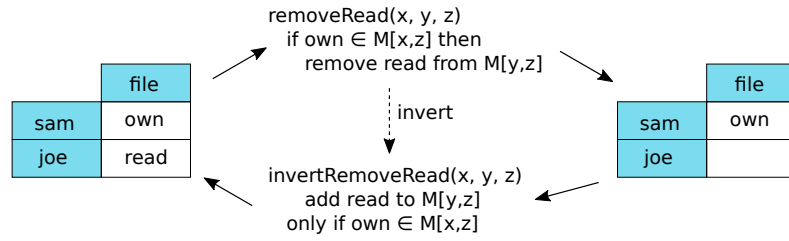
Safety analysis is motivated by delegation [77]: the power that users, who are not fully trusted,

possess to change authorizations. Delegation, in turn, is motivated by the need to scale access control systems – relying on only a few fully trusted users to change authorizations does not scale to large numbers of users and resources. If indeed an instance of such safety analysis reveals that an undesirable state may be reachable in future, we may want to change the current authorization policy, or the nature of changes that users who are not fully trusted are allowed to make to authorizations, so that the possibility of reaching such an undesirable state is precluded.

Safety analysis is part of security through prevention; that is, ensuring that our system can never be insecure. Unfortunately, while prevention is certainly part of an overall approach to securing systems, it is widely acknowledged that it is often insufficient [78]. There are two reasons for this. One is that it is not always possible to characterize the set of undesirable states that we want to preclude a system from reaching. Another is that even if we are able to characterize such a set, the underlying analysis problem may be intractable, or even undecidable [65].

A complementary approach to security via prevention, is security via detection – detecting, after the fact, that a breach has occurred. Action can then be taken, such as finding the perpetrator, and holding them accountable. Also, this may give us insight into preventive mechanisms to preclude similar breaches in the future. Forensics is part of security via detection. Questions we pose in the context of forensic analysis relate to whether past states of a system indicate a violation, relative to our current mindset, i.e., after the fact. Thus, from the standpoint of access control, forensic analysis is a natural complement of safety analysis: safety asks about the future, forensics asks about the past (see Figure 4.1).

**Forensics in Access Control** We argue that access control is an important and useful context in which to pose and consider forensic analysis. The reason is, as we mention above, access control is a critical part of the security of a system. Therefore, an ability to answer questions about past authorizations can provide important clues and evidence regarding breaches. Alternately, and equally as importantly, it may provide evidence for exoneration – reasonable doubt as to one’s guilt. These themes recur in the next section, in which we pose the forensics analysis problem in access control. To our knowledge, we are the first to do so.



**Figure 4.2.** A natural way to ask whether a past state may have existed is to first “invert” the state-change rule. The figure shows an example for an access control system based on the HRU scheme [24]. In “inverting” a state-change, a pre-condition becomes a post-condition. In the example, we would specify the arguments  $x = \text{sam}$ ,  $y = \text{joe}$ ,  $z = \text{file}$  to go from one state to the other.

As we discuss further there, we can think of three kinds of basic questions one may ask about past authorization states.

One is whether a user, say, Alice, may have possessed a particular privilege. For example, whether she may have been able to write to a file. Another is whether she indeed possessed such a privilege. A third is whether she exercised a privilege. Of course, the third implies the second, which in turn implies the first: if indeed she exercised a privilege, then we know that she possessed it, and if she possessed it, then we know that she may have possessed it.

One may ask why we care about the first two kinds of questions at all. The answer to this lies in our intent. A natural way to motivate the weakest of the above questions, i.e., whether Alice may have held a privilege, is to consider its complement — whether it is impossible that Alice held a privilege. An answer of ‘true,’ to this complementary question, can be used to exonerate Alice of a crime. A weaker query may also suffice if an investigator seeks clues, and not definitive evidence.

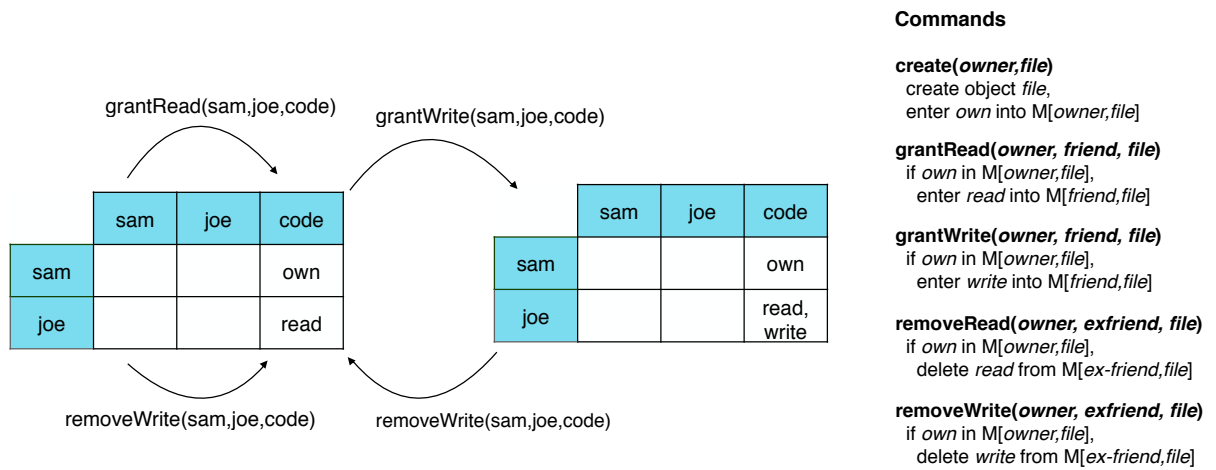
**Differences from safety** Notwithstanding the apparent correspondence between safety and forensic analysis, there are some important differences. A difference is that a natural way to intuit an algorithm for forensic analysis in a particular setting is to first “invert” state-change rules and then check whether a past state with a certain property may have existed. It is customary in access control, and indeed, more broadly in computing, to describe a system by specifying a current state, and the manner in which a state can change to a new state. This is

exactly what we do in the specification of a Turing machine, for example [6].

An example from access control is shown in Figure 4.2. The state in the example is encoded as an access matrix – the rows are subjects and columns are objects; in the example, we have one object only, *file*, and two subjects, *sam* and *joe*. A manner in which a state can change is specified as a command in the Harrison-Ruzzo-Ullman (HRU) scheme [65]. In the example, the command allows a subject who has *own* access to *file* to remove *read* access from a subject. Thus, this command contains a precondition – the check for whether the former subject indeed has *own* access. If we were to think of the past from the new state then, we presumably would consider the “inverse” of the command. A natural way to specify such an “inverse” is using a post-condition as shown in the figure. We leverage this mindset in devising an algorithm for forensic analysis in a Role-Based Access Control (RBAC) scheme called Administrative RBAC (ARBAC) [79] (see the proof for Theorem 4.3.5 in Section 4.3.2).

Another difference between forensic and safety analysis is that in an instance of safety analysis, we typically specify two states: a current state, and a future “goal” state. The latter may be specified partially only, for example, whether a particular subject possesses a particular right over a particular object. We ask whether such a “goal” state is reachable from the current state. In forensic analysis, apart from the current state and a past “goal” state, it seems meaningful to specify also a known state from the past that delimits how far back we look for the “goal” state. Such a state is shown to the far left as a double circle in Figure 4.1. The reason for this is that as we are looking in to the past, a superuser who runs the forensic analysis may already know of such a state, and by explicitly specifying it, is able to tailor the analysis so they are able to zero-in on the answer to their forensic question. For example, we may be aware that the security incident we are investigating happened within the past two weeks, and so it is meaningful to specify a known past state from two weeks ago to delimit our search. Furthermore, it is typical for a superuser to first configure a system to first be in some state, and only then delegate the ability to change the authorization state to potentially untrustworthy users. The superuser, in their forensic analysis, may want to delimit forensic analysis with such a state. In our specification of forensic analysis in access control in Section 4.2, we allow the explicit specification of such a state. We point out, however, that such a state can be an “empty” state that signifies the “start of time” for that access control system. That is, we can





**Figure 4.3.** An example of an access control system in the HRU scheme [65].

completely relax this additional constraint, if needed.

A third difference between forensic and safety analysis is that as forensics considers past events, at least some of those events may have been recorded in logs. These logs, in turn, can be an additional input to forensic analysis.

**Logs** Logs are data we record and store about past states of a system. They are useful for security based on after-the-fact detection. In Section 4.4 we identify that indeed, incorporation of prudently recorded logs can make the forensic analysis problem computationally easier. This may not seem surprising, but the associated challenge here is that indiscriminate logging can result in inordinately large logs that we need to store, perhaps over long periods of time, before we pose a forensic analysis instance. A hypothesis of this work, which we investigate further in Section 4.4, is that depending on our intent, we can do *goal-directed* logging; that is, log only those things that are potentially useful for forensic analysis in future. Our goal can significantly reduce the amount of logs we collect and have to store over a possibly long period of time before we pose a forensic analysis instance in response to, for example, a suspicion that a breach occurred at some time in the past. Thus, goal-directed logging can be seen as a kind of semantic, lossy, compression technique for logs. We also address the issue of sufficient and minimal logs in Section 4.4.

The issue of sufficient logs is tied closely to the kind of forensic question we seek to ask. Also, it relates to a fundamental technical challenge with both safety and forensic analysis: the possibility of several, even infinitely many, reachable states. We illustrate this using the example in Figure 4.3 which is a system in the Harrison-Ruzzo-Ullman (HRU) scheme [65]. As the figure shows, the scheme uses an access matrix to encode a state, and a set of commands as the state-change rule (see Section 4.3.1 for more details). Suppose our current state is the left state in the figure in which *joe* does not have *write* to *code*, and the known past state which delimits our search is the same state. Now suppose we seek to know: did *joe* ever possess *write* to *code*? Without logs, it would be impossible to know the answer. A sufficient log would be a bit attached to *joe* that is set if and only if *joe* gets *write* to *code*. Also sufficient are logs on every action by the owner *sam*. Now suppose we relax the question to: could *joe* have possessed *write* to *code*? The answer to this question may be true or false, again based on the logs we maintain. Suppose, we maintain no logs whatsoever. Then the answer is ‘true.’ Suppose, instead, we maintain as logs all actions by the owner *sam*. Then the answer is ‘true’ or ‘false’ based on whether *sam* did indeed grant *write* to *joe*.

**Foundations** We formulate the problem in Section 4.2; to our knowledge, we are the first to pose the forensic analysis problem in access control as such. We then make two sets of contributions at the foundations. In Section 4.3, we consider three access control schemes from the literature: (i) the HRU scheme [65], (ii) the Role-Based Access Control (RBAC) scheme, Administrative RBAC (ARBAC) [79], and (iii) the Graham-Denning scheme [80]. We identify the computational complexity of forensic analysis for each of those schemes. Our reason for considering these schemes is that they are qualitatively different from one another: HRU is based on the access-matrix and commands that fire, ARBAC is based on RBAC, and Graham-Denning is a discretionary scheme. Also, the computational complexity of safety analysis is known for each of these schemes, and they belong to different classes: for HRU, it is undecidable [65], for ARBAC, it is **PSPACE**-complete [81], and for Graham-Denning, it is in **P** [71]. We identify that for each of these schemes, forensic analysis, in the worst-case, lies in the same class as safety analysis, thereby establishing a tight correspondence between safety and forensics in this regard as well. Our other set of contributions at the foundations is with sufficient and minimal logs. In Section 4.4 we study goal-directed logging to reduce the

size of logs needed to maintain precision and improve efficiency of forensic analysis for these schemes.

**Practice** We have conducted a case-study on forensic analysis in the access control system of a practical application called Hello, Retail! [82]. Hello, Retail! is a serverless cloud application [83], intended to be run in the Amazon Web Services (AWS) public cloud [84]. We chose Hello, Retail! as it is open-source, has won a serverless architecture award [85], serverless cloud applications are considered to be at the cutting-edge of application development and deployment and a security policy configuration has been published with Hello, Retail!’s code. We simulated a security incident and used logging that AWS supports to carry out forensic analysis. Our study and results are in Section 4.5.

**Other sections** In addition, we discuss related work in Section 4.6, and conclude with Section 4.7.

## 4.2 Problem

As we mention in Section 4.1, an access control system can be perceived as a state-change system. This perspective is consistent with prior work, particularly as it relates to safety analysis (see, for example, [65, 74, 70, 81, 75]). We first recall this characterization in Section 4.2.1, and then, in Section 4.2.2, pose the forensic analysis problem in a manner that is meaningful for any access control scheme. In Section 4.2.3 we explain how the notion of ‘trusted’ users can be incorporated in the problem formulation.

### 4.2.1 Access Control Schemes and Systems

An access control scheme is a pair  $\langle \Gamma, \Psi \rangle$ , where  $\Gamma$  is a set of states and  $\Psi$  is a set of state-change rules. In practice,  $\Gamma$  may be infinite, but  $\Psi$  is finite. A state,  $\gamma \in \Gamma$ , encodes the rights or privileges that a subject has at a given time. That is, it contains all the information necessary to make an access control decision at that point in time. Examples of subjects are

users and roles. An access control request can be seen as encoded by a *query* which is issued on a state. The syntax of a query is custom to a particular scheme; in the next section in which we instantiate concrete schemes, we adopt specific queries for each. For example, in Figure 4.3, we consider one resource only, and we can perceive a query as a pair,  $\langle \text{subject}, \text{right} \rangle$ , which encodes the query as to whether *subject* possesses *right* over the resource. When a query  $q$  is issued on a state  $\gamma \in \Gamma$ , it evaluates to either true, which we denote  $\gamma \vdash q$ , or false, which we denote  $\gamma \not\vdash q$ . For example, in Figure 4.3, the state to the left  $\vdash \langle \text{joe}, \text{read} \rangle$ , but the state to the left  $\not\vdash \langle \text{joe}, \text{write} \rangle$ .

A state-change rule,  $\psi \in \Psi$ , specifies how a state may change to another. Each such  $\psi \in \Psi$  is a function which maps a state, and possibly additional arguments, to another state. For example, a particular  $\psi$  may specify that a state in which a subject  $a$  which possesses a right *own* on a resource, and a subject  $b$  which does not possess a right to that resource can change to a state in which  $a$ 's privileges remain unchanged, but  $b$  has the right to the resource. This is exactly the state change from the state to the left of Figure 4.3 to the state on the right.

Given two states  $\gamma$  and  $\gamma'$  and a state-change rule  $\psi$ , we write  $\gamma \rightarrow_\psi \gamma'$  if the change from  $\gamma$  to  $\gamma'$  is allowed by  $\psi$ , and  $\gamma \rightarrow_\psi^* \gamma'$  if a sequence of zero or more allowed state-changes leads from  $\gamma$  to  $\gamma'$ . For example, in Figure 4.3, left state  $\rightarrow_\psi^*$  right state, and left state  $\rightarrow_\psi^*$  left state.

An access control system based on a scheme  $\langle \Gamma, \Psi \rangle$  is an instance of that scheme, i.e.,  $\langle \gamma, \psi \rangle$ , where  $\gamma \in \Gamma$ , is the current state, and  $\psi \in \Psi$ , is the state-change rule for that system. Figure 4.3 illustrates an access control system based on the HRU scheme [65].

**Three assumptions** We make three assumptions regarding state-changes in an access control system. One is that at any moment in time, at most one state-change can be in execution. That is, we do not allow for state-changes in parallel. Another assumption is that a state-change that is enabled either occurs successfully, or does not occur at all. We do not have a notion of partial state-changes. The third assumption is that whenever we consider a sequence of state-changes,  $\gamma \rightarrow_\psi^* \gamma'$ , it is finite. All these assumptions have been adopted in prior work in the context of safety analysis, and articulated explicitly [72]. The motivation for these assumptions is that they model practice.

## 4.2.2 Forensic Analysis: Problem Formulation

Given an access control scheme  $\langle \Gamma, \Psi \rangle$ , a forensic instance is  $\langle \gamma, \psi, k, q, \pi, L \rangle$ , where (we discuss each component in more detail farther below):

- $\langle \gamma, \psi \rangle$  is an access control system, i.e.,  $\gamma \in \Gamma$  is the current state and  $\psi \in \Psi$  is the state-change rule;
- $k$  is a known past state that delimits which past states we consider;
- $q$  is a query, which encodes the kind of state we seek to find;
- $\pi \in \{\text{possible}, \text{actual}\}$  encodes whether we are asking if a state in which  $q$  is true may have occurred, or did occur;
- $L$  is a log.

**Comparison to safety analysis** Before we discuss each of the above components in more detail, we compare the above formulation to the manner in which safety analysis has been expressed in prior work (see, for example, [65, 70, 74, 75, 81]). An instance of safety analysis is  $\langle \gamma, \psi, q \rangle$  – given the system  $\langle \gamma, \psi \rangle$ , we ask whether there exists a future state in which  $q$  is true. Thus, in forensic analysis, in addition to those components, we consider  $k, \pi$  and  $L$ . These are discussed in Section 4.1; we discuss them below in more detail. Some work in safety analysis, for example, that of Jha et al. [81], also incorporates a set of trusted users who are assumed to not effect any state-changes. We can certainly incorporate such a set in our formulation of forensic analysis. We address this in Section 4.2.3.

**The system,  $\langle \gamma, \psi \rangle$**  As with safety analysis, we assume that we are given an access control system, i.e., a current-state  $\gamma$  and the state-change rule  $\psi$ . We seek to explore past states going backwards from  $\gamma$  under the assumption every “forward” state-change meets  $\psi$ .

**A known past state,  $k$**  The state  $k$  delimits the state-space in our search. That is, we explore only those states that lie between the current state,  $\gamma$ , and  $k$ . There are two reasons we include

such a component in our problem-formulation. One is that for many practical instances, the initial state of the system may not be an “empty” one. Indeed, it is typical for a superuser to create users and objects and assign certain rights to the users over the objects before initializing the access control system. Secondly, the known past state also allows us to limit how far back in time our analysis should go. We assume that  $\gamma$  is reachable from  $k$ , i.e.  $k \rightarrow_{\psi}^* \gamma$ .

**The query,  $q$**  The query characterizes the kind of state we seek in our state-space search. While the exact syntax of a query is custom to the particular access control scheme, in this work, just as in safety analysis, we adopt as query an access request. For example, in a scheme based on an access matrix, a query is  $\langle \text{subject, object, right} \rangle$ . As another example, in an RBAC scheme, a query may be  $\langle \text{user, role} \rangle$ . Safety analysis has been generalized to so-called security analysis [69, 71] in which more general queries are considered. Certainly, we can generalize forensic analysis to such queries as well; we leave that for future work.

**Possible vs. actual,  $\pi$**  In Section 4.1, we discuss three kinds of forensic questions that seem meaningful in the context of access control: whether a user, say, Alice, (i) may have possessed a privilege, (ii) did possess a privilege, and, (iii) exercised a privilege. From the perspective of state-reachability, (i) and (ii) are both meaningful to pose. Analysis type (iii) pertains to access enforcement, rather than authorization of an access, and is beyond the scope of the kind of forensic analysis we consider. The component  $\pi$  is intended to capture whether we mean type (i) possible, or type (ii) actual.

**Log,  $L$**  A log,  $L$ , is a record of past states and state-changes; it is a sequence of entries, each of which is a portion of a state or state-change that has occurred. A log may not be comprehensive: not only may an entry contain only a portion of a state or state-change, but also, not all states and state-changes that have occurred may appear in a log. For example, only some, and not all authorizations that are part of a state, may have been recorded as an entry.

We can think of the log as a ground truth. With this perspective, we can attempt to establish properties of interest to us in forensics using logical entailment [86]. We may establish, for example, given  $\langle \psi, L \rangle$ , where  $\psi$  is the state-change rule and  $L$  the log from a forensic analysis instance, that a state-change sequence,  $\gamma_1 \rightarrow_{\psi} \gamma_2 \rightarrow_{\psi} \dots \rightarrow_{\psi} \gamma_n$ , for  $n \geq 1$ , did indeed occur.

We may establish, instead, that a given state-change sequence  $\gamma_1 \rightarrow_\psi \dots \rightarrow_\psi \gamma_n$  did not occur. It is possible, of course, that a statement we pose cannot be established, and neither can its complement. For example, for a particular sequence  $\gamma_1 \rightarrow_\psi \dots \rightarrow_\psi \gamma_n$ , we may be able to neither establish that it occurred, nor that it did not occur.

**Semantics of an instance** Given a forensic analysis instance, it can be assigned a truth-value of one of true, false or unknown. If  $\pi = \text{possible}$ , then the instance cannot be unknown, and is either true or false. It is true if and only if there exists a state  $\gamma'$  such that: (1)  $\gamma' \vdash q$ , and, (2) there exists a sequence  $k \rightarrow_\psi^* \gamma' \rightarrow_\psi^* \gamma$  for which we are not able to establish that it did not occur. It is false otherwise. We clarify that Condition (2) is that such a sequence may indeed have occurred; we are unable to exclude that possibility.

If  $\pi = \text{actual}$ , the output may be true, unknown or false. It is true if and only if there exists a state  $\gamma'$  such that: (1)  $\gamma' \vdash q$ , and, (2) we are able to establish that some sequence  $k \rightarrow_\psi^* \gamma' \rightarrow_\psi^* \gamma$  did indeed occur. It is false if and only if we are able to establish that in the sequence  $k \rightarrow_\psi^* \gamma$  that occurred in the past, there is no state  $\gamma'$  such that  $\gamma' \vdash q$ . Otherwise, it is unknown.

We are careful with the above characterizations so that they are logically consistent with one another. For example, if a forensics instance with  $\pi = \text{actual}$  is true, then that instance is guaranteed to be true if we change  $\pi$  to  $\text{possible}$ . It is the case also that if an instance with  $\pi = \text{possible}$  is false, then that instance remains false if we change  $\pi$  to  $\text{actual}$ .

An investigator may set  $\pi$  to  $\text{possible}$  or  $\text{actual}$  depending on their intent. If their intent is, for example, exoneration of a suspect, then  $\text{possible}$  may be the more appropriate of the two, with the hope that the instance is false. If their intent is establishment of guilt,  $\text{actual}$  may be more appropriate, with the hope that the instance is true.

In Section 4.3, we instantiate the above ideas for concrete access control schemes. We also present examples of concrete systems and forensics instances for those schemes.

### 4.2.3 Trusted Users

In the safety problem one asks if a state in which an undesirable access is authorized can be reached in the future. Such a state is deemed to be unsafe. Prior work on safety analysis (e.g. [65, 70, 81]) consider the notion of trusted users. These are users who are assumed to make secure choices when they change the system's authorization policy. Thus, a safety instance that includes a set of trusted users,  $T$ , in its input, asks if an unsafe state can be reached in future via state-changes that are effected by users not in  $T$ , i.e., untrusted users. To clarify, it may be possible for users in  $T$  to effect state-changes that result in an unsafe state in future, but these users are trusted not to do so.

Our formulation of forensic analysis in the previous subsection can be adapted to include a set of trusted users,  $T$ . The way to do this, however, needs careful consideration. As discussed in Section 4.2.2, in forensics, the path of interest,  $k \rightarrow_{\psi}^* \gamma' \rightarrow_{\psi}^* \gamma$ , is specified by 3 states.  $k$  and  $\gamma$  are the input known past and current states respectively.  $\gamma'$  is the goal state that the analysis seeks to find, i.e.,  $\gamma' \vdash q$ , where  $q$  is the input query. As in safety analysis, the goal state in forensic analysis can be considered 'unsafe'. Thus, it is reasonable for forensics to be concerned only with sequences  $k \rightarrow_{\psi}^* \gamma'$  in which the state-changes are effected by untrusted users only. An observation is that if the goal state in a forensics instance is a safe state, there is no reason to specify a set of trusted users in the input. As for the sequence  $\gamma' \rightarrow_{\psi}^* \gamma$ , if  $\gamma$  is unsafe, then the same rule applies, i.e., forensics is interested in sequences in which state-changes are effected by untrusted users only. If  $\gamma$  is safe, then the state-changes can be effected by any kind of user, trusted or untrusted.

**Semantics of an instance** We now explain the semantics of a forensic instance that includes a set of trusted users,  $T$ . We assume  $\gamma$  is safe; our formulation requires minor modifications only for the case in which it is not. Given a forensic instance  $\langle \gamma, \psi, k, q, \pi, L, T \rangle$ , if  $\pi = \text{possible}$ , the instance is either true or false. It is true if and only if there exists a state  $\gamma'$  such that: (1)  $\gamma' \vdash q$ , (2) there exists a sequence of state-changes  $k \rightarrow_{\psi}^* \gamma'$  effected by users not in  $T$ , (3) there exists a sequence of state-changes  $\gamma' \rightarrow_{\psi}^* \gamma$  and, (4) we are not able to establish that the sequences of (2) or (3) did not occur.



If  $\pi = \text{actual}$ , the output may be true, unknown or false. It is true if and only if there exists a state  $\gamma'$  such that: (1)  $\gamma' \vdash q$ , and, (2) we are able to establish that some sequence  $k \rightarrow_{\psi}^* \gamma' \rightarrow_{\psi}^* \gamma$  did indeed occur. It is false if and only if we are able to establish that in the sequence  $k \rightarrow_{\psi}^* \gamma$  that occurred in the past, there is no state  $\gamma'$  such that  $\gamma' \vdash q$ . Otherwise, it is unknown. Observe that for  $\pi = \text{actual}$ , we make no mention of state-changes being effected by trusted vs. untrusted users. This is because if  $\gamma' \vdash q$  and  $k \rightarrow_{\psi}^* \gamma'$  actually happened, then by the definition of trusted users, the state-changes must have been effected by untrusted users.

In the remainder of this chapter, we do not consider the notion of trusted users as it has no consequence to our results.

## 4.3 Computational Complexity

In this section, we analyze the worst-case computational complexity of the forensic analysis problem posed in the previous section for three access control schemes from the literature. An intent is to determine whether efficient algorithms exist for forensic analysis in these access control schemes, in the worst-case. Also of interest is whether forensic analysis has a different worst-case computational complexity than safety analysis for these schemes.

**Proof strategy** We adopt customary approaches to establish both lower- and upper-bound computational hardness. To establish lower-bound computational hardness, we reduce from a problem that is known to have that lower-bound computational hardness. To establish an upper-bound, we do so by construction: we propose an algorithm. In the proofs for Theorems 4.3.1 and 4.3.3 in Sections 4.3.1 and 4.3.2 respectively, we choose to reduce from the safety analysis problem for that particular scheme. This not only establishes the desired lower-bound, but also establishes a correspondence between safety and forensics for these schemes.

### 4.3.1 HRU

In this section, we analyze the worst-case computational complexity of forensic analysis for the HRU scheme [65]. We first describe the scheme, present an example system and forensic analysis instances for it, and then state and prove that forensic analysis for the scheme is undecidable in the worst-case.

**The HRU scheme** We assume the existence of two sets, subjects  $\mathbb{S}$  and objects  $\mathbb{O}$  with  $\mathbb{S} \subseteq \mathbb{O}$ . Every system is associated with a finite set of rights,  $R$ . A state,  $\gamma$ , is associated with a subset of subjects and objects that exist in that state, and an access matrix that maps each subject-object pair to a subset of the set of rights that that subject possesses over that object. We write  $M_\gamma[s, o]$  for the entry, or cell, in the access matrix in the state  $\gamma$ , that is the set of rights the subject  $s$  possesses over the object  $o$ . To refer to the access matrix in its entirety, we write  $M_\gamma[]$ .

A state-change rule  $\psi$  in the HRU scheme is a set of commands,  $C_\psi$ . A state-change is the firing of a command in  $C_\psi$ . Each command takes arguments, each of which is a subject or object, has an optional precondition, which is a conjunction of checks for the presence of a right in a cell for a subject and object that are arguments, and a sequence of primitive operations. The HRU scheme specifies six primitive operations: (i) the creation of a subject, which creates a new row and column in the access matrix, (ii) the creation of an object which is not a subject, which creates a new column, (iii) the deletion of a subject, which removes the corresponding row and column, (iv) the deletion of an object that is not a subject, which removes the corresponding column, (v) the addition of a right to a cell, if it is not already in the cell, and, (vi) the removal of a right from a cell, if the cell contains that right.

Figure 4.3 shows an example of an HRU system in which the set of rights  $R = \{own, read, write\}$  and  $C_\psi$  comprises five commands. We refer the reader to prior work [65, 72] for a more comprehensive description of the HRU scheme.

**Forensic analysis in HRU** In a forensic analysis instance  $\langle \gamma, \psi, k, q, \pi, L \rangle$  for an HRU system,  $\gamma$  and  $k$  are access matrices, and  $\psi$  is a set of commands such that  $k \rightarrow_\psi^* \gamma$ , and a

corresponding sequence of state-changes did indeed occur.  $L$ , the log, comprises portions of states and state-changes; it is empty if logging is not enabled. A query  $q$  corresponds to an access request, which, for an access matrix, is a triple  $\langle s, o, r \rangle$  where  $s$  is a subject,  $o$  is an object and  $r$  is a right.

**Example – possible access** Consider Figure 4.3 and suppose we ask if it is possible that *sam* had *read* access to *code* in a past state, and we have no information about the path taken from the initial state (left state in Figure 4.3) to the current state (right state in Figure 4.3). The query is true because the following is a possible sequence of commands: *grantRead(sam, sam, code)*, *removeRead(sam, sam, code)*, *grantWrite(sam, joe, code)*. Now suppose we happened to have logged every state-change, and the log tells us that the only state-change that occurred is *grantWrite(sam, joe, code)*. Then the answer to the query is false.

**Example – actual access** Consider Figure 4.3 and suppose we ask if *sam* had *read* access to *code* in a past state, and we have no information about the path taken from the initial state to the current state. The instance is unknown because it may be true or false depending on the path taken. If we know from our logs that the only command that fired is *grantWrite(sam, joe, code)*, then the instance is false.

**Computational complexity** We now state the worst-case computational complexity of forensic analysis in the HRU scheme in the following theorem.

**Theorem 4.3.1.** *Forensic analysis for the HRU scheme is undecidable in the worst-case.*

Before we prove Theorem 4.3.1 that characterizes the worst-case computational complexity of forensic analysis in the HRU scheme, we recall the particular kind of safety analysis from prior work that we leverage in our proof of that theorem. The safety question is: given that a particular subject  $s$  does not possess a particular right  $r$  over a particular object  $o$  in the current state, does there exist a reachable state in which it does? Then, we identify the computational complexity of this safety problem for a restricted version of the HRU scheme. To our knowledge, this result is new to our work; however, the proof is directly from prior work [72].

**Definition 1** ( $((s, o, r)$ -simple safety [72]). *Given an HRU-system, with the current state  $\gamma$ , a*

state-change rule  $\psi$ , a right  $r \in R$ , an object  $o \in \mathbb{O}$  and a subject  $s \in \mathbb{S}$ , the corresponding  $(s, o, r)$ -simple-safety instance is true if and only if (1) either  $r \notin M_\gamma[s, o]$ , or at least one of  $s, o$  does not exist in  $\gamma$ , and there exists a state  $\gamma'$  such that: (2)  $\gamma \rightarrow_\psi^* \gamma'$ , and, (3) both  $s, o$  exist in  $\gamma'$ , and  $r \in M_{\gamma'}[s, o]$ .

**Theorem 4.3.2.**  $(s, o, r)$ -simple safety for an HRU system in which we disallow the primitive operations: (i) destroy subject, and, (ii) destroy object, is undecidable.

*Proof.* Prior work [72] proves that  $(s, o, r)$ -simple-safety is undecidable for the HRU scheme in general. We observe that that proof employs a reduction which outputs an HRU system, none of whose commands contains a destroy subject or destroy object primitive operation.  $\square$

*Proof of Theorem 4.3.1.* We adopt the class of forensic analysis instances  $\langle \gamma, \psi, k, q, \pi, L \rangle$  in which  $\pi = \text{possible}$ ,  $\gamma \not\models q$  and  $L = \emptyset$ . We then reduce from the  $(s, o, r)$ -simple-safety problem for an HRU system in which the primitive operations that destroy objects and subjects are absent. Recall that such a safety instance takes as part of its input a triple  $\langle s, o, r \rangle$ . Underlying our proof is the observation that the forensic instance with  $q$  set to the  $\langle s, o, r \rangle$  from the safety instance is true if and only if both the following are true:

**Condition 1:** there exists a state  $\gamma_1$  such that  $s, o$  exist in  $\gamma_1$ ,  $r \in M_{\gamma_1}[s, o]$  and  $k \rightarrow_\psi^* \gamma_1$ , and,

**Condition 2:** there exists a state  $\gamma_2$  such that  $r \notin M_{\gamma_2}[s, o]$ , and  $\gamma_1 \rightarrow_\psi^* \gamma_2$ , and  $\gamma_2 \rightarrow_\psi^* \gamma$

The reduction uses the following intuition. The input safety instance is mapped onto Condition 1, and the current state of the forensic instance is chosen so that by defining appropriate HRU commands, if Condition 1 is satisfied, then Condition 2 is satisfied.

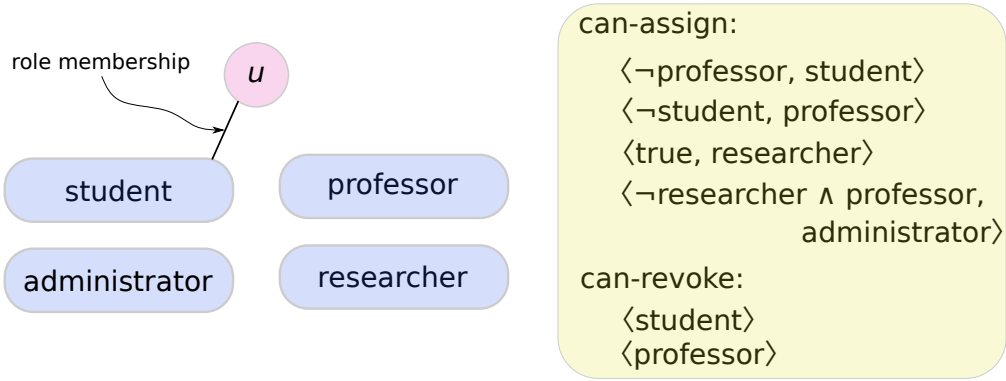
The reduction is as follows. Let the safety instance be  $\langle \alpha_s, \psi_s, q_s \rangle$ , where  $\alpha_s$  is the current state,  $\psi_s$  is the state-change rule, and  $q_s = \langle s^*, o^*, r^* \rangle$  is the query. Let the forensic instance we would like to map to be  $\langle \gamma, \psi, k, q, \pi = \text{possible}, L = \emptyset \rangle$ . Then the parameters of the forensic instance are set as follows:

- $q = q_s$

- $k$  is such that the subjects and objects that exist in  $k$  are the subjects and objects that exist in  $\alpha_s$  plus a new subject,  $s_{new}$  and a new object  $o_{new}$ , and  $M_k$  is  $M_{\alpha_s}$  with a row and column for  $s_{new}$ , and a column for  $o_{new}$  added.
- The set of rights is the same as from the safety instance, plus a new right,  $r_{new}$ .
- $\gamma$  is such that the only subject that exists is  $s_{new}$ , the only object that exists other than  $s_{new}$  is  $o_{new}$ , and the access matrix is such that  $M_\gamma[s_{new}, o_{new}] = \{r_{new}\}$  and  $M_\gamma[s_{new}, s_{new}] = \emptyset$ .
- $\psi$  is  $\psi_s$  plus three new commands  $Remove_{r^*}$ ,  $Destroy_s$  and  $Destroy_o$  which are specified below.

$Remove_{r^*}(s_1, s_2, o_1, o_2)$  checks if  $r^* \in M[s_1, o_1]$  and if yes, it removes it and enters  $r_{new}$  into  $M[s_2, o_2]$ .  $Destroy_s(s_1, s_2, o_1)$  checks if  $r_{new} \in M[s_1, o_1]$  and if yes, it destroys  $s_2$ .  $Destroy_o(s_1, o_1, o_2)$  checks if  $r_{new} \in M[s_1, o_1]$  and if yes, it destroys  $o_2$ . These three commands collectively ensure that once Condition 1 is satisfied (i.e.  $s^*$  possesses  $r^*$  over  $o^*$ ), then Condition 2 can be satisfied by calling  $Remove_{r^*}(s^*, s_{new}, o^*, o_{new})$ , and then the commands  $Destroy_o(s_{new}, o_{new}, o_2)$  and  $Destroy_s(s_{new}, s_2, o_{new})$  as many times as necessary, each time with the appropriate parameters for  $o_2$  and  $s_2$ , to reach the current state  $\gamma$ . Hence it suffices to show that the safety instance is true if and only if the forensic instance satisfies Condition 1.

Assume the safety instance is true. This means there exists a state  $\beta$  that is reachable from the current state,  $\alpha_s$ , of the instance, such that  $r^* \in M_\beta[s^*, o^*]$ . Since the known past state of the forensic instance,  $k$ , is derived from  $\alpha_s$ , and the forensic instance includes all of the commands of the safety instance, this implies that the forensic instance satisfies Condition 1. For the other direction, assume Condition 1 can be satisfied for the resulting forensic instance. We observe that the sequence of commands that can be used to get from  $k$  to  $\gamma_1$  need not contain any of the commands  $Remove_{r^*}$ ,  $Destroy_s$  or  $Destroy_o$ . This is because in HRU, the condition of a command checks only for the presence of rights, and hence removing the right  $r^*$  from a cell, or destroying subjects or objects, is not beneficial in fulfilling the conditions of other commands. Furthermore, adding  $r_{new}$  to a cell also is not beneficial because none of



**Figure 4.4.** An ARBAC system with four roles, four can-assign rules in  $CA_\psi$ , and two can-revoke rules in  $CR_\psi$ . In the state,  $\gamma$ , shown,  $UA_\gamma = \{\text{student}\}$ .

commands from the safety instance check for  $r_{new}$  in their condition. Hence, if Condition 1 can be satisfied for the forensic instance, then it can also be satisfied using only the commands that were part of the safety instance, which implies the safety analysis instance is true.  $\square$

We observe also that even though we have adopted an empty log,  $L = \emptyset$ , in the above proof, there exist classes of instances in which  $L \neq \emptyset$  for which the problem remains undecidable. An example of such a class is one in which the contents of  $L$  are irrelevant to the forensic question.

### 4.3.2 ARBAC

We now consider an access control scheme, ARBAC [79], which is role-based. As in the previous section, we first describe ARBAC, provide examples, and then identify the worst-case computational complexity of forensic analysis in ARBAC.

**The ARBAC scheme** We adopt and describe a restricted version of the ARBAC scheme that suffices for our lower-bound in Theorem 4.3.3. As with safety, our results carry over to the general version [87]. In the ARBAC scheme, a state is based on RBAC. In our restricted version, a state is the set of roles to which a particular user, whom we associate with the access

control system as part of the input, is assigned. That is, each access control system in ARBAC is associated with a set of roles  $R$  which does not change across states. There is a particular user,  $u$ , who also is part of the specification of the system and does not change across states. A state,  $\gamma$ , is  $UA_\gamma$  where  $UA_\gamma \subseteq R$  is the set of roles to which the user  $u$  is assigned in the state  $\gamma$ . The different subsets of  $R$  are the different possible states.

A state-change rule,  $\psi$ , with which a system is associated comprises two sets of rules: can-assign rules,  $CA_\psi$ , and can-revoke rules,  $CR_\psi$ . Each member of  $CA_\psi$  is a pair  $\langle \text{precondition}, \text{target role} \rangle$ . Each member of  $CR_\psi$  is a role. A precondition in a member of  $CA_\psi$  is either the mnemonic ‘true,’ or a conjunction of roles or their negation. E.g., given the set of roles  $R = \{r_1, \dots, r_{50}\}$ , a precondition in a rule in  $CA_\psi$  may be  $r_1 \wedge \neg r_2 \wedge \neg r_{23}$ . The target role in a member of  $CA_\psi$  is a role from  $R$ , e.g.,  $r_{15}$ . The meaning of a rule  $\langle \text{precondition}, \text{target role} \rangle$  is that if the role-membership of the user  $u$  in our system satisfies the precondition, then  $u$  may be assigned to the target role. In our example, if  $u$  is a member of  $r_1$ , and a member of neither  $r_2$  nor  $r_{23}$ , then  $u$  may be assigned to  $r_{15}$ .

The meaning of a member, which is a role, in  $CR_\psi$  is that the user  $u$ ’s membership from that role may be revoked. A state-change is the firing of a rule in  $CA_\psi$  or  $CR_\psi$ , i.e., either the assignment of the user  $u$  to a role, or the revocation of  $u$  from a role. Thus, if we transition from a state  $\gamma$  to  $\gamma'$ , then  $UA_{\gamma'}$  is either  $UA_\gamma \cup \{r\}$ , if the state-change was the assignment of  $u$  to  $r$ , or,  $UA_\gamma \setminus \{r\}$ , if the state-change was the revocation of  $u$  from  $r$ . In the former case, we assume that  $r \notin UA_\gamma$ , and in the latter,  $r \in UA_\gamma$ . That is, a state-change must change  $u$ ’s role-membership.

Figure 4.4 shows an example of an ARBAC system. It has four roles and its state-change rule  $\psi$  comprises the  $CA_\psi$  and  $CR_\psi$  that is shown in the figure. We show also a state,  $\gamma$ , in which  $u$  is a member of one of the four roles.

**Forensic analysis in ARBAC** In a forensic analysis instance,  $\langle \gamma, \psi, k, q, \pi, L \rangle$  in ARBAC, the states  $\gamma$  and  $k$  are subsets of  $R$  and are the roles to which the user  $u$  in the system is assigned to in those states. The state-change rule,  $\psi$ , comprises the two sets  $CA_\psi$  and  $CR_\psi$ . The log,  $L$ , is portions of states and state-changes. As before,  $L$  may be empty. A query,  $q$ , is a role,  $\langle r \rangle$ . The meaning of a query is: is the user  $u$  with which the system is associated, a

member of  $r$ ?

**Example – possible access** Suppose our system is the one shown in Figure 4.4, and we have a forensic instance in which  $\gamma$ , the current state, and  $k$ , the known past state, are both the state shown in the figure, i.e.,  $u$  is assigned to the role student only. Suppose we adopt  $\pi = \text{possible}$ ,  $L = \emptyset$  and the query  $q = \langle \text{professor} \rangle$ ; that is, we are asking whether it is possible that in a past state,  $u$  was a member of the role professor. This instance is true. We can intuit this by going “backwards” starting at  $\gamma$  and ending at  $k$ . A possible such sequence of state-changes is: *assign-to(student)*, *revoke-from(professor)*, *assign-to(professor)*, *revoke-from(student)*. Now suppose  $L$  is every state that occurred, or, alternately, every state-change that occurred, starting at  $k$ . And from  $L$ , we observe that  $u$  was never revoked from student. Then the instance is false.

**Example – actual access** Consider the system in Figure 4.4, and let  $\gamma = k$  be the state shown in the figure. Suppose  $\pi = \text{actual}$ ,  $L = \emptyset$  and  $q = \langle \text{professor} \rangle$ . Then the forensics instance is unknown because it may or may not have been the case that  $u$  was a member of *professor*. If, on the other hand,  $L$  records every state-change, or even every exercise of a rule in  $CA_\psi$ , then the instance is true if there is an assignment to the role *professor* in  $L$ , and false otherwise.

**Computational complexity** We state our lower-bound in Theorem 4.3.3 below. The proof is similar to the proof for Theorem 4.3.1, and relies on the fact that safety analysis for the version we consider is **PSPACE**-complete. **PSPACE** is the class of decision problems for each of which there exists an algorithm that consumes space at worst polynomial in the size of the input [6].

**Theorem 4.3.3.** *Forensic analysis for ARBAC is **PSPACE**-hard in the worst-case.*

Before we prove Theorem 4.3.3, we revisit safety analysis for the particular restricted version of ARBAC that we adopt. Shalen et al. [88] observe that safety analysis for the same kind of query as we adopt for forensic analysis is **PSPACE**-complete, which is the same class as to which more general versions of ARBAC belong. As that observation is made somewhat in passing in that work and we rely on it for our proof of Theorem 4.3.3, we call it out clearly in



the following theorem.

**Theorem 4.3.4.** *Safety analysis for the version of ARBAC we adopt, with a query = a role, as we adopt, is **PSPACE**-complete.*

*Proof.* As prior work [88] observes, the reduction of Jha et al. [81] for **PSPACE**-hardness maps to only those instances that belong to our restricted version. The upper-bound of **PSPACE** remains as our version is less general.  $\square$

We now present the proof for Theorem 4.3.3.

*Proof of Theorem 4.3.3.* In our forensic analysis instance,  $\langle \gamma, \psi, k, q, \pi, L \rangle$ , we adopt  $\pi = \text{possible}$ ,  $L = \emptyset$  and  $\gamma \not\models q$ . We reduce from safety analysis for the version of ARBAC we discuss above, in a manner similar to our proof for Theorem 4.3.1. Let the safety instance be  $\langle R_s, UA_s, CA_s, CR_s, q_s = \langle r^* \rangle \rangle$ , where  $R_s$  is the set of roles in the instance,  $UA_s \subseteq R_s$  is the set of roles to which the user is assigned in the current state,  $CA_s, CR_s$  comprise the state-change rule and  $q_s = \langle r^* \rangle$ , where  $r^* \in R_s$ , is the query. That is, whether there exists a future state in which the user is a member of  $r^*$ . We map the safety instance to a forensics analysis instance  $\langle \gamma, \psi, k, q, \pi = \text{possible}, L = \emptyset \rangle$  as follows.

1. The set of roles in the system for our forensics instance is the same as in the safety instance,  $R_s$ ;
2. The set of roles to which the user is assigned in the known past state,  $UA_k = UA_s$ ;
3. The query  $q = q_s = \langle r^* \rangle$ ;
4. The set of roles to which the user is assigned in the current state,  $UA_\gamma = R_s \setminus \{r^*\}$ ; and,
5. The state-change rule,  $\psi$  comprises  $CA_\psi = CA_s \cup \{\langle r^*, r \rangle : r \in R_s \setminus \{r^*\}\}$  and  $CR_\psi = CR_s \cup \langle r^* \rangle$ .

We now claim that the safety instance is true if and only if the following two conditions hold for the forensics instance:

**Condition 1:** there exists a state  $\gamma_1$ , such that  $\gamma_1 \vdash q$ , and  $k \rightarrow_{\psi}^* \gamma_1$ .

**Condition 2:** there exists a state  $\gamma_2$ , such that  $\gamma_2 \not\vdash q$ ,  $\gamma_1 \rightarrow_{\psi}^* \gamma_2$ , and  $\gamma_2 \rightarrow_{\psi}^* \gamma$ .

As in the proof for Theorem 4.3.1, in our reduction, we map the input safety instance to Condition 1, and the current state,  $\gamma$ , of the forensic instance is chosen so that by defining appropriate  $CA_{\psi}$  and  $CR_{\psi}$  rules, if Condition 1 is satisfied, then Condition 2 is satisfied. We first prove this: that Condition 1 implies Condition 2. Suppose the forensic instance meets Condition 1. This means that there exists a state  $\gamma_1$  such that  $\gamma_1 \vdash q$ . Now, to reach state  $\gamma_2$  as above, we exercise each rule in  $CA_{\psi}$  that is not in  $CA_s$  as specified in (5) above. This results in the user being assigned to every role in  $R_s$ . We then exercise new rule in  $CR_{\psi}$  to revoke the user from  $r^*$ . Thus, Condition 2 is satisfied.

It remains to be shown that the safety instance is true if and only if Condition 1 is met. For the “only if” direction, assume the safety instance is true. Then, we observe that  $CA_{\psi} \supseteq CA_s$  and  $CR_{\psi} \supseteq CR_s$ . Thus, every state-change in the safety instance can be effected in the forensic instance to go from the state  $k$  to the state  $\gamma_1$ . Thus, Condition 1 is true.

For the “if” direction, assume the forensic instance satisfies Condition 1. We claim via the following reasoning that the members in the set  $CR_{\psi}$  that do not exist in  $CR_s$ , and the members in the set  $CA_{\psi}$  that do not exist in  $CA_s$  do not affect whether Condition 1 holds. This in turn implies that the safety instance is true.

Our reasoning is as follows.  $\langle r^* \rangle \in CR_{\psi}$  allows for the revocation of the user from  $r^*$ . The revocation cannot happen unless Condition 1 is already satisfied, i.e., unless the user is already a member of  $r^*$ . And,  $\langle r^*, r \rangle \in CA_{\psi}$  allows for the user to be assigned to any role in  $R_s$ , provided that the user is a member of  $r^*$ , i.e., Condition 1 is already satisfied. Hence, Condition 1 can be satisfied by the forensic instance as a result of members in  $CA_{\psi} \cap CA_s$  and  $CR_{\psi} \cap CR_s$  which implies that the safety instance is true.  $\square$

The above theorem gives us a worst-case lower-bound for forensic analysis in the ARBAC scheme. The following theorem gives us an upper-bound. However, we restrict it in the following ways. (i) We allow  $\pi = \text{possible}$  only. (ii) We set the log,  $L = \emptyset$ . (iii) We

consider our restricted version of ARBAC only. The reason for (i) is that that is the only version which is a decision problem, i.e., each instance is associated with true/false only. The reason for (ii) is that without a more precise specification of what exactly  $L$  contains, we cannot meaningfully analyze it. The reason for (iii) is that that is the version to which we restrict ourselves in the proof for Theorem 4.3.3 above. For (ii), in the next section, we discuss sufficient logs that render forensic analysis efficient. For (iii), we conjecture that our upper-bound remains even if we adopt more general versions of ARBAC. Our basis for this conjecture is the strong correspondence between safety and forensics, and the fact that the computational complexity of safety analysis is not impacted by the generalizations to ARBAC that have been considered in the literature. In Appendix A we describe a general version of ARBAC considered in prior work on safety analysis [81] and show that forensic analysis for it has the same upper-bound as the restricted version we consider here.

**Theorem 4.3.5.** *Forensic analysis for the ARBAC scheme we adopt with  $\pi = possible$  and  $L = \emptyset$  is in **PSPACE**.*

*Proof.* Our proof is by construction. We propose a non-deterministic algorithm that is correct and is guaranteed to halt, and that consumes space at worst polynomial in the size of the input, thereby proving that the problem is in **NPSpace**, the class of decision problems for which there exists a non-deterministic algorithm that consumes space only polynomial in the size of the input. We then appeal to Savitch’s theorem, **NPSpace** = **PSPACE** [7]. Given a forensics instance,  $\langle \gamma, \psi, k, q, \pi, L \rangle$ , our algorithm “inverts” state-changes, and goes “backwards” starting at the current state,  $\gamma$ .

We maintain a “present state,” call it  $p$ . (We call it “present” to distinguish from “current state” which is an input.) As  $p$  is a subset of the set of roles, it can be represented by  $|R|$  bits, which is polynomial in the size of the input. We initialize  $p$  to  $\gamma$ . We then enumerate all possible state-changes that could have resulted from an immediately prior state,  $p'$  to  $p$ . This is our “inversion” of state-changes. There can be at most linearly, and therefore, polynomially, many such state-changes because the number is upper-bounded by the size of  $\psi$ , i.e.,  $|CA_\psi| + |CR_\psi|$ .

To enumerate these possible state-changes, we do the following. For each  $\langle c, r \rangle \in CA_\psi$ , we ask whether  $r \in UA_p$ , i.e., the user in the system is a member of  $r$  in the state  $p$ . If yes, we ask

whether the precondition  $c$  is satisfied by  $UA_p \setminus \{r\}$ . If yes, then that member of  $CA_\psi$  was a possible state-change, with the prior state  $p' = UA_p \setminus \{r\}$ . Otherwise, it is not a possible state-change. Similarly, for each  $\langle r \rangle \in CR_\psi$ , we ask whether  $r \in UA_p$ . If yes, then we know that that member  $\langle r \rangle$  of  $CR_\psi$  was not a possible state-change. Otherwise, it was with the prior state  $p' = UA_p \cup \{r\}$ .

Then, from amongst the possible state-changes, we non-deterministically pick one, and set  $p$  to the corresponding prior state,  $p'$ . Apart from space for enumerating possible state-changes and space for  $p$ , we allocate space also for: (i) the number of “backwards” state-changes we have effected, and, (ii) a boolean for whether we have encountered a state  $p$  such that  $p \vdash q$ . The size of (i) is  $|R|$  bits, where  $R$  is the set of all roles. This is because we have at most  $2^{|R|}$  possible different states, and therefore a counter of  $\log 2^{|R|} = |R|$  bits suffices. The size of (ii) is one bit.

Every time we effect a (non-deterministic) state-change backwards with a new present state  $p$ , we do the following. We increment our counter (i) above. Then we check whether the bit (ii) above is already set. If yes, we check whether  $p = k$ , the known past state. If yes, we halt and output ‘true.’ If the bit (ii) is not set, we check whether the new state  $p \vdash q$ . If yes, we set the bit (ii) and we clear the state counter to 0. If we do not return ‘true’ after this state-change, then we check whether our counter has reached  $2^{|R|}$ . If yes, we immediately halt and output ‘false.’ Otherwise, we continue, i.e., enumerate possible state-changes with the new present state  $p$ .

The algorithm above is correct because there exists a sequence of non-deterministic choices that causes us to output ‘true’ if and only if the input instance is indeed true. Also, the algorithm is guaranteed to halt because of the counter we maintain. The total space we need is at worst linear, and therefore polynomial, in the size of the input.  $\square$

**Corollary 4.3.6.** *Forensic analysis for the ARBAC scheme we adopt with  $\pi = possible$  and  $L = \emptyset$  is **PSPACE**-complete.*

**Observations** As with the HRU scheme, forensic analysis in ARBAC, at least for the version we consider for possible instances with no logs, is in the same complexity class as

safety analysis. Unlike the HRU scheme, however, it is decidable, and the proof for Theorem 4.3.5 can be seen as constructive: it provides an algorithm, albeit a non-deterministic one. There is considerable prior work on tools, in practice, for safety analysis of ARBAC and its variants (see, for example, [87, 89, 90, 91]). Underlying these tools is a reduction to variants of model-checking, each complete for **PSPACE**. Given that forensic analysis for ARBAC is in **PSPACE**, we expect that such tools can be effectively re-purposed for forensic analysis.

### 4.3.3 Graham-Denning

We now consider a third scheme from the literature, the Graham-Denning scheme [80]. We consider it for two reasons. One is that it is qualitatively different from both the HRU and ARBAC schemes: it is a discretionary scheme, i.e., there is a notion of ownership of objects by subjects, and the granting of rights based on a subject's discretion. The other reason is that similar to HRU and ARBAC, safety analysis has been studied for Graham-Denning [70]. We first reproduce the description of the scheme from prior work [70], provide some examples, and then identify the worst-case computational complexity of forensic analysis in Graham-Denning. The proof is constructive: it is an algorithm. It is similar to the algorithm for safety analysis from prior work [70] in that it is a case-analysis.

**The Graham-Denning Scheme** The authorization state in Graham-Denning can be encoded using an access matrix, exactly as in the HRU scheme, with one exception. And that is that there is a special subject, call it a superuser, which exists in every state. The set of rights can be partitioned into three sets: (i) an arbitrary set of rights associated with a system like in the HRU scheme, call it  $R$ , (ii) a set of rights, denote it  $R^*$ , which are the transferrable versions of every right in  $R$ , i.e.,  $R^* = \{r^* : r \in R\}$ ; the semantics of an  $r^*$  are that a user who possesses  $r^*$  over an object may grant  $r$  or  $r^*$  to another user over that object, and, (iii) two distinguished rights *own* and *control* neither of which is in  $R$  nor  $R^*$ . The state-change rule is fixed for every instance of the Graham-Denning scheme, and is a set of commands. This set, reproduced from prior work [70] is shown in Figure 4.5.

A state must satisfy seven properties; the state-change rule, i.e., the commands, maintain these

properties in the next state.

1. Every object must be owned by at least one subject.
2. Only subjects are controlled.
3. The special subject  $u$  exists in the state. It is not owned by any subject, and it is not controlled by any other subject.
4. A subject other than  $u$  is owned by exactly one other subject (a subject cannot own itself).
5. Every subject controls itself.
6. A subject other than  $u$  is controlled by at most one other subject.
7. There exists no set of subjects that form a cycle in terms of ownership of each other.

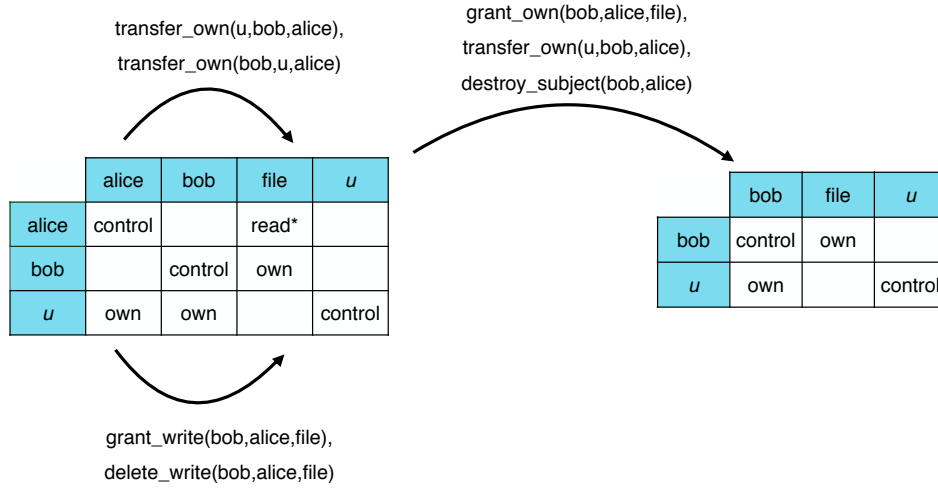
Figure 4.6 shows an example of a Graham-Denning system in which, besides the special subject  $u$ , there are two subjects, *alice* and *bob*, and one object, *file*. The figure shows two states, and examples of sequences of commands from Figure 4.5 that may have transpired in going from one state to the other.

**Forensic analysis in Graham-Denning** In a forensic analysis instance  $\langle \gamma, \psi, k, q, \pi, L \rangle$  for a Graham-Denning system,  $\psi$  is the set of commands shown in Figure 4.5, and  $\gamma$  and  $k$  are access matrices such that  $k \rightarrow_{\psi}^* \gamma$ , and a corresponding sequence of state-changes did indeed occur.  $L$ , the log, comprises portions of states and state-changes; it is empty if logging is not enabled. A query  $q$  corresponds to an access request which is a triple  $\langle s, o, x \rangle$  where  $s$  is a subject,  $o$  is an object and  $x$  is a right.

**Example - possible access** Suppose our system is the one shown in Figure 4.6, and we have a forensic instance in which  $\gamma$ , the current state, and  $k$ , the known past state that delimits our search, are both the left state in the figure, and we ask if it is possible that *alice* had *write* to *file* in some state between  $k$  and  $\gamma$ . If the log,  $L$ , is empty, then the query is ‘true’ because the

$\text{command transfer}_r(i, s, o)$ $\text{if } r^* \in M_\gamma[i, o] \wedge s \in S_\gamma \text{ then}$ $M_{\gamma'}[s, o] \leftarrow M_\gamma[s, o] \cup \{r\}$	$\text{command transfer}_{r^*}(i, s, o)$ $\text{if } r^* \in M_\gamma[i, o] \wedge s \in S_\gamma \text{ then}$ $M_{\gamma'}[s, o] \leftarrow M_\gamma[s, o] \cup \{r^*\}$
$\text{command transfer\_own}(i, s, o)$ $\text{if } \text{own} \in M_\gamma[i, o] \wedge o \in S_\gamma \wedge s \in S_\gamma \text{ then}$ $\text{if } \nexists \{s_1, \dots, s_n\} \in S_\gamma \text{ such that}$ $\text{own} \in M_\gamma[s_1, s] \wedge \text{own} \in M_\gamma[s_2, s_1]$ $\wedge \dots \wedge \text{own} \in M_\gamma[s_n, s_{n-1}]$ $\wedge \text{own} \in M_\gamma[o, s_n] \text{ then}$ $M_{\gamma'}[s, o] \leftarrow M_\gamma[s, o] \cup \{\text{own}\}$ $M_{\gamma'}[i, o] \leftarrow M_\gamma[i, o] - \{\text{own}\}$	$\text{command grant}_r(i, s, o)$ $\text{if } \text{own} \in M_\gamma[i, o] \wedge s \in S_\gamma \text{ then}$ $M_{\gamma'}[s, o] \leftarrow M_\gamma[s, o] \cup \{r\}$
$\text{command grant\_control}(i, s, o)$ $\text{if } \text{own} \in M_\gamma[i, o] \wedge o \in S_\gamma \wedge s \in S_\gamma \text{ then}$ $\text{if } \nexists s' \in S_\gamma \text{ such that}$ $s' \neq o \wedge \text{control} \in M_\gamma[s', o] \text{ then}$ $M_{\gamma'}[s, o] \leftarrow M_\gamma[s, o] \cup \{\text{control}\}$	$\text{command grant}_{r^*}(i, s, o)$ $\text{if } \text{own} \in M_\gamma[i, o] \wedge s \in S_\gamma \text{ then}$ $M_{\gamma'}[s, o] \leftarrow M_\gamma[s, o] \cup \{r^*\}$
$\text{command delete}_r(i, s, o)$ $\text{if } (\text{own} \in M_\gamma[i, o] \wedge s \in S_\gamma)$ $\vee \text{control} \in M_\gamma[i, s] \text{ then}$ $M_{\gamma'}[s, o] \leftarrow M_\gamma[s, o] - \{r\}$	$\text{command delete}_{r^*}(i, s, o)$ $\text{if } (\text{own} \in M_\gamma[i, o] \wedge s \in S_\gamma)$ $\vee \text{control} \in M_\gamma[i, s] \text{ then}$ $M_{\gamma'}[s, o] \leftarrow M_\gamma[s, o] - \{r^*\}$
$\text{command create\_object}(i, o)$ $\text{if } o \notin O_\gamma \wedge i \in S_\gamma \wedge o \in \mathcal{O} - \mathcal{S} \text{ then}$ $O_{\gamma'} \leftarrow O_\gamma \cup \{o\}$ $M_{\gamma'}[i, o] \leftarrow \text{own}$	$\text{command destroy\_object}(i, o)$ $\text{if } \text{own} \in M_\gamma[i, o] \wedge o \notin S_\gamma \text{ then}$ $O_{\gamma'} \leftarrow O_\gamma - \{o\}$
$\text{command create\_subject}(i, s)$ $\text{if } s \notin O_\gamma \wedge i \in S_\gamma \wedge s \in \mathcal{S} \text{ then}$ $O_{\gamma'} \leftarrow O_\gamma \cup \{s\}$ $S_{\gamma'} \leftarrow S_\gamma \cup \{s\}$ $M_{\gamma'}[i, s] \leftarrow \{\text{own}\}$ $M_{\gamma'}[s, s] \leftarrow \{\text{control}\}$	$\text{command destroy\_subject}(i, s)$ $\text{if } \text{own} \in M_\gamma[i, s] \wedge s \in S_\gamma \text{ then}$ $\forall o \in O_\gamma, \text{if } \text{own} \in M_\gamma[s, o] \text{ then}$ $M_{\gamma'}[i, o] \leftarrow M_\gamma[i, o] \cup \{\text{own}\}$ $O_{\gamma'} \leftarrow O_\gamma - \{s\}$ $S_{\gamma'} \leftarrow S_\gamma - \{s\}$

**Figure 4.5.** The set of commands that constitutes the state-change rule,  $\psi$ , for a system based on the Graham-Denning scheme. The first parameter  $i$  is the initiator of the command. There is one  $\text{transfer}_r$ ,  $\text{grant}_r$ , and  $\text{delete}_r$  command for each  $r \in R_b$ , and one  $\text{transfer}_{r^*}$ ,  $\text{grant}_{r^*}$  and  $\text{delete}_{r^*}$  command for each  $r^* \in R_b^*$ . Figure reproduced from [70].



**Figure 4.6.** An example of an access control system in the Graham-Denning scheme [80].

following is a possible sequence of commands: *grant\_write(bob, alice, file)*, *delete\_write(bob, alice, file)*.

**Example - actual access** Given the same current and past known states as the example above, suppose we now ask if *alice* actually had *write* to *file* in the past. If the log,  $L$ , is empty, then the query is ‘unknown’ because there exists a path between  $k$  and  $\gamma$  such that the query is true (e.g. *grant\_write(bob, alice, file)*, *delete\_write(bob, alice, file)*), and there also exists a path between  $k$  and  $\gamma$  such that the query is false (e.g. *transfer\_own(u,bob,alice)*, *transfer\_own(bob,u,alice)*). If the log,  $L$ , is not empty and records all state-changes, then the instance is true if there is a state-change in  $L$  that grants or transfers *write* over *file* to *alice*, and it is false otherwise.

**Computational Complexity** We first recall the result for safety analysis from prior work [70]. That work adopts queries of the same form as we adopt, i.e.,  $q = \langle s, o, x \rangle$ , and proves that safety analysis is in **P** for the Graham-Denning scheme. The proof is by construction: that work devises an algorithm and proves that it is correct and runs in polynomial-time in the worst-case. The algorithm is a case-analysis: as a simple example, if in the query  $q = \langle s, o, x \rangle$ , the right  $x = \text{control}$  and the object  $o$  is not also a subject, then by Property (2) in the list of properties above, we can immediately conclude that no reachable state exists that satisfies  $q$ .



Not all cases are as straightforward.

In the below theorem, we establish that like safety, forensic analysis for Graham-Denning with  $\pi = \text{possible}$ ,  $L = \emptyset$  and  $q$  of the form  $\langle s, o, x \rangle$ , is tractable.

**Theorem 4.3.7.** *Forensic analysis in the Graham-Denning scheme with  $\pi = \text{possible}$ ,  $L = \emptyset$  and  $q$  of the form  $\langle s, o, x \rangle$ , is in **P**.*

*Proof.* We consider the computational complexity of only those forensic instances,  $\langle \gamma, \psi, k, q, \pi, L \rangle$ , where  $q = \langle s, o, x \rangle$ , that satisfy  $\pi = \text{possible}$ ,  $L = \emptyset$ . We adopt the same proof-strategy as prior work on safety [70], and propose an algorithm that carries out a case-analysis. Thus, we prove by construction that the problem lies in **P**, which, as with the HRU and ARBAC schemes, is the same class in which safety analysis lies. Algorithm 3 is our algorithm.  $\square$

**Input:** Forensic instance,  $\{\gamma, \psi, k, q = \langle s, o, x \rangle, \text{possible}, \emptyset\}$

**Output:** true or false

- 1 **if**  $\gamma \vdash q \vee k \vdash q$  **then** output true
- 2 **if**  $x = \text{control}$  **then**
- 3     **if**  $o \notin \mathbb{S}$  **then** output false
- 4     **if**  $o \in S_\gamma \wedge s \in S_\gamma$  **then** output false
- 5     **if**  $o \in S_k \wedge \exists s' \in S_k \text{ s.t. } s' \neq o \wedge \text{control} \in M_k[s', o] \wedge s' \in S_\gamma$  **then** output false
- 6 **if**  $x = \text{own}$  **then**
- 7     **if**  $o \in O_\gamma \setminus S_\gamma \wedge s \in S_\gamma$  **then** output false
- 8 output true

**Algorithm 3:** Algorithm for forensic analysis for the Graham-Denning scheme when  $\pi = \text{possible}$  and  $L = \emptyset$ .  $\mathbb{S}$  denotes the set of subjects in the scheme.

Following is a line-by-line explanation of Algorithm 3.

**[Line 1]** If the query is true in the current state  $\gamma$  or the known past state  $k$ , the instance is true.

**[Line 2,3]** If  $x = \text{control}$ , and  $o$  is not a subject, the instance is false because only subjects are controlled.

**[Line 2,4]** If  $x = \text{control}$ , and  $o$  and  $s$  are subjects that exist in the current state  $\gamma$ , the instance is false because  $\gamma \not\models q$  (see Line 1) and the *control* right cannot be revoked or transferred (see commands in Figure 4.5).

**[Line 2,5]** If  $x = \text{control}$ , and  $o$  is a subject that exists in the known state  $k$ , and  $o$  is controlled by some subject  $s'$  in  $k$ , besides itself, and  $s'$  also exists in  $\gamma$ , then the instance is false. This is because a subject can be controlled by at most one other subject besides itself, and the *control* right cannot be revoked or transferred.

**[Line 2,8]** If  $x = \text{control}$ , and none of the checks in lines 3-5 output false, then the instance is true because  $o$ 's owner can grant *control* over  $o$  to  $s$  in some state between the states  $k$  and  $\gamma$ .

**[Line 6,7]** If  $x = \text{own}$ , and if  $o$  is not a subject, and  $o$  and  $s$  both exist in  $\gamma$ , the instance is false, because when a subject owns an object that is not a subject, this ownership is retained until either the subject or the object are destroyed, and  $\gamma \not\models q$  (see Line 1).

**[Line 6,8]** If  $x = \text{own}$ , and the check in line 7 does not output false, then the instance is true, because if  $o$  is a subject, its owner can transfer ownership to  $s$ , and if  $o$  is not a subject, its owner can grant ownership to  $s$ . Figure 4.6 shows examples for these two cases. Suppose the past known state  $k$  and the current state  $\gamma$  are both identical to the state to the left of the figure, it is possible that *bob* had *own* to *alice* in the past because a possible sequence of commands that may have executed between  $k$  and  $\gamma$  is *transfer\_own(u,bob,alice)*, *transfer\_own(bob,u,alice)*. Suppose that  $k$  and  $\gamma$  are the states to the left and right of the figure, respectively, it is possible that *alice* had *own* to *file* in the past because a possible sequence of commands is *grant\_own(bob,alice,file)*, *transfer\_own(u,bob,alice)*, *destroy\_subject(bob,alice)*.

**[Line 8]** If  $x \in R \cup R^*$ , then the instance is true, because the owner of  $o$  can grant  $x$  to  $s$  and then revoke  $x$  from  $s$ .

Algorithm 3 does not account for the possibility of trusted users. In Appendix B we present a forensic analysis algorithm for Graham-Denning based systems that considers a set of trusted

users.

**Observations** The class **P** can be seen as an upper-bound for the computational complexity of forensic analysis in the Graham-Denning scheme. Thus, our proof strategy is akin to our proof for Theorem 4.3.5 in the previous section: we propose an algorithm. Thus, for all three schemes we consider, forensic analysis lies in the same class as safety analysis.

## 4.4 Goal-directed Logging

We now consider logging. Logs record information about past states, and can aid forensics. In this work, we assume that the logs are trustworthy. For example, they may be recorded on write-once media. If the logging mechanism is comprehensive, that is, it logs every state-change that occurs such that all prior states between the initial and current state can be reconstructed from the logs, then forensic analysis as it was posed in Section 4.2 becomes tractable. One would just have to scan the log entries linearly beginning from the initial state in order to decide if the given query was true in a past state. However, there are significant drawbacks associated with this kind of comprehensive logging. Firstly, the size of the log may explode. Secondly, it results in the *quantity problem* [92]— a large amount of data that needs to be analyzed in a forensic investigation. This problem contributes significantly to the time needed to complete the analysis. Thirdly, logging irrelevant data may obstruct the accuracy of the forensic analysis by distracting the analyst from the relevant data [93].

Data reduction techniques can be used to control the size of logs that need to be analyzed. For example, in the investigation of a computer hard disk, files that are known to be of no interest to the investigation from an examiner’s viewpoint can be eliminated through the use of file filters. In the context of access control, a way to reduce the amount of information logged is to use *goal-directed logging*. That is, if we know in advance or restrict the types of forensic analysis that can arise, then we need to log only the necessary information to answer the analyses that are of interest.

We point out that what we mean by goal-directed logging is different than what has been

proposed as *goal-oriented logging* [94, 93]. In prior work, the term goal-oriented logging has been used to suggest that logging requirements can be extracted from analyzing the goals of an attacker. This is done by considering the security policies in place and analyzing the ways in which an attacker can violate these. In this work, we do not attempt to derive logging requirements from security policies because there is no guarantee that such policies achieve perfect security. Indeed there may be situations in which the deficiency of security policies leads to a breach. In this case, if the logs are extracted solely from the policies, they would not contain the information needed to expose or analyze such a breach.

Of course, the goals of a forensic analysis may not be known in advance. In this case, although goal-directed logging cannot be used to reduce the size of the logs that are recorded, it can be used to reduce the size of the logs that are input to the analysis. For an access control scheme, a *minimal log* contains exactly the information required to answer the particular type of forensic analysis with true or false, and no information more than that. If any part of a minimal log is missing, then for *possible* analysis, there exists a query such that the analysis is undecidable, and for *actual* analysis there exists a query such that the answer is unknown. A *sufficient log* contains all the information that a minimal log provides, but may also contain more information than that. The following theorem follows immediately from the fact that a log that is sufficient for *actual* analysis is sufficient for *possible* analysis.

**Theorem 4.4.1.** *For any access control scheme, the size of a minimal log for possible analysis is less than or equal to the size of a minimal log for actual analysis.*

**HRU** In the absence of logs, *possible* analysis in HRU is undecidable (see Section 4.3.1). Let the number of commands, objects and rights, in a given HRU access control system be  $C$ ,  $B$  and  $R$ , respectively. We first consider some candidate logs.

Suppose our log consists of a counter that increments every time a command is executed successfully. Let the number of commands that have been executed successfully be denoted by  $x$ . Then the size of the log is  $\approx \log_2 x$  assuming binary encoding, because that is the size we need for a counter. (In the sentence immediately prior, the first use of “log” refers to the record, and the second refers to logarithm.) Let the maximum number of arguments in any command of the system be  $j$ . Then given a forensic analysis instance  $\langle \gamma, \psi, k, q, \pi, L \rangle$ , we can

explore all the paths from  $k$  to  $\gamma$  in  $O((C \cdot B^j)^x)$  operations, which is super-exponential in the size of the forensic instance.

As another candidate, suppose our log consists of a list of the commands executed successfully, but without their arguments. Now, we cannot simply maintain a counter of size  $\log_2 x$ , but rather, if  $x$  is the number of command executions, the log has  $x$  entries, and the size of the log is  $x \cdot \log_2 C$ . In this case we can explore all the relevant paths from  $k$  to  $\gamma$  in  $O((B^j)^x)$  operations, which is an improvement, but still exponential in the size of the instance.

As a final candidate, suppose our log consists of a list of the commands executed successfully, with their arguments. Without any loss of generality, assume that every command has  $j$  parameters. The size of the log is now  $x \cdot (\log_2 C + j \cdot \log_2 B)$ . This log allows us to reconstruct the exact path taken from the initial state to the current state, in linear time in the size of the instance.

**Theorem 4.4.2.** *Given an access control system in the HRU scheme, a size of a sufficient log for a polynomial-time algorithm for possible analysis instances is  $(x - y) \cdot (\log_2 C + j \cdot \log_2 B)$ , where  $y = \min(c, x)$  and  $c$  is a constant.*

*Proof.* The log consists of a list of all but a constant number,  $y$ , of the commands that are executed successfully with their arguments. Also assume that the commands not logged are the ones that are executed first. That is, the logging mechanism only starts logging once the number of commands,  $x$ , is at least  $y$ . The size of the log is  $(x - y) \cdot (\log_2 C + j \cdot \log_2 B)$ . Given such a log, we can explore all relevant paths from  $k$  to  $\gamma$  in  $O((C \cdot B^j)^y)$  operations which is polynomial in the size of the input forensic instance.  $\square$

**Theorem 4.4.3.** *In the worst-case, for an access control system in the HRU scheme, the lower-bound on the size of a minimal log for a polynomial-time algorithm to output either ‘true’ or ‘false’, and not ‘unknown’, to an actual analysis instance is  $x \cdot (\log_2 C + j \cdot \log_2 B)$ .*

*Proof.* For actual analysis, the log must allow for the reconstruction of the exact path taken from initial to the current state. Therefore, in the worst-case, the log must consist of a list of all the commands executed successfully, with their arguments. Figure 4.3 in Section 4.1 depicts

an example showing how even when a single command is not logged, it can be uncertain whether a query is true or false. Suppose that the known past state and the current state are identical to the left state in the figure, and the log records only the second of two commands that are fired and this is *removeWrite(sam,joe,code)*. From the figure we see that it is not possible to answer whether *joe* actually had *write* access to *code* without knowing what the first command fired was. Even when a single parameter is not logged, it can be uncertain whether a query is true or false. For an example that demonstrates this, consider Figure 4.3 and suppose that the known past state and current state are identical to the left state in the figure but in addition to *own*,  $M[sam, code]$  also contains *write*. Now suppose two commands have fired and the log records *grantWrite(sam,x,code),removeWrite(sam,joe,code)* where *x* denotes the missing parameter. In this case also, it is not possible to answer whether *joe* actually had *write* access to *code* as it depends on whether the missing parameter is *sam* or *joe*.  $\square$

**ARBAC** In the absence of logs, *possible* analysis for the ARBAC scheme is **PSPACE**-complete (see Section 4.3.2). In the following, let the number of distinct preconditions that appear in the *CA* state component of the access control system be *C* and let the set of roles be *R*.

**Theorem 4.4.4.** *Given an access control system in the ARBAC scheme, the size of a sufficient log, for a polynomial-time algorithm for possible analysis instances is  $\min(C, |R|)$ .*

*Proof.* If  $\min(C, |R|) = C$  we maintain a bit for each precondition where the bit is set if the precondition is ever satisfied by the role assignment of the user *u* to whom the system is associated with. Then given a *possible* query,  $q = \langle r \rangle$ , we can answer it efficiently by using the following algorithm. First check if *u* is assigned to *r* in the past known or current states, and if yes, output true. Next, linearly scan *CR* to check if *r* can be revoked, and if not output false. Otherwise linearly scan *CA* and output true if and only if there exists a member  $\langle c, r \rangle \in CA$  such that the precondition bit for *u* associated with *c* is set. If  $\min(C, |R|) = |R|$ , we maintain a bit for each role in *R*, where the bit is set if *u* is ever assigned to the role. Then given a *possible* query,  $q = \langle r \rangle$ , we output true if the bit associated with *r* is set, otherwise we output false. This algorithm runs in time linear in the size of the forensic instance.  $\square$

**Theorem 4.4.5.** *In the worst-case, for an access control system in the ARBAC scheme, the lower-bound on the size of a minimal log for a polynomial-time algorithm that outputs either ‘true’ or ‘false’, and not ‘unknown’, for `actual` analysis instances is  $|R|$ .*

*Proof.* In the proof of Theorem 4.4.4 above, we explain how logs of size  $|R|$  can be used to answer a query efficiently. We now reason why in the worst-case such a sized log is minimal. Assume that we reduce the log by one bit. That is, we do not log whether or not the user  $u$  in the system and query was assigned to a particular role  $r^*$ . We can construct a forensic instance for which the resulting log is insufficient. The instance has the following properties: (1) the query is  $\langle r^* \rangle$ , (2) the query is not true in the initial and current states, (3)  $\langle r^* \rangle \in CR$ , i.e.,  $u$  can be revoked from  $r^*$ , and, (4) there is only one  $\langle c, r \rangle \in CA$  such that  $r = r^*$ , and for this element,  $c = \text{true}$ . For such an instance, the log reduced by one bit is not sufficient to answer if  $u$  was definitively assigned to  $r^*$  in a past state because that is exactly the information we excluded from the log, and there is no other way to get this information.  $\square$

**Graham-Denning** In the absence of logs, forensic instances in the Graham-Denning scheme in which  $\pi = \text{possible}$  is in **P** (see Section 4.3.3). Thus, we can state the following theorem regarding logs for `possible` analysis instances.

**Theorem 4.4.6.** *The size of a sufficient log for a polynomial-time algorithm for `possible` forensics analysis instances for the Graham-Denning scheme is 0.*

For `actual` analysis, we provide a lower-bound in the following theorem.  $C$  and  $B$  are the number of commands and objects, respectively, in the system. As the proof states, it relies directly on our result for the HRU scheme above. This, as it turns out, is a benefit of adopting the syntax for the HRU scheme for encoding the Graham-Denning scheme.

**Theorem 4.4.7.** *In the worst-case, for an access control system in the Graham-Denning scheme, the lower-bound on the size of a minimal log for a polynomial-time algorithm that outputs either ‘true’ or ‘false’, and not ‘unknown’, for `actual` analysis instances is at least  $x \cdot (\log_2 C + 2 \cdot \log_2 B)$ .*

*Proof.* For `actual` analysis, the log must allow for the reconstruction of the exact path taken from initial to the current state. Therefore in the worst-case the log must consist of a list of all the commands executed successfully, with their arguments. The minimum number of arguments in a command is two, and the maximum is three (see Figure 4.5). The examples for HRU in the proof for Theorem 4.4.3 can be adapted to be examples for Graham-Denning of when if a single command or parameter is not logged, we are forced to output ‘unknown’. □

**Observations** If we assume that in ARBAC, the number of distinct preconditions is smaller than the number of roles in the system, then the log-size that suffices for `possible` analysis is smaller than the minimal logs that are needed for `actual` analysis in the worst-case, for all of the three schemes we consider. An interesting consequence of this relates to goal-directed logging: if we know that our objective is `possible` analysis only, then we can make do with smaller log-size. This observation, in turn, validates the goal-directed mindset whose intent is one of reducing the log-size, or rendering analysis easier by more quickly trimming down the logs to only what is needed.

## 4.5 Case Study

We now discuss a case-study we have conducted on a real software system to validate some of our analytical insights from the previous sections. In our case study, we first deploy an open source application called Hello, Retail! [82] on the AWS cloud platform [84]. We then simulate a security incident. The scope of our study was to answer the following questions about AWS: (1) What types of forensic queries does it support? (2) Can goal-directed logging help to reduce the size of logs? In Section 4.5.1, we provide some background on the portion of AWS’s access control model on which we focus, the logging service it provides on which we focus, and the Hello, Retail! application. In Section 4.5.2 we describe the security incident that we simulate and our observations.

**Limitations** Our case-study is not a comprehensive assessment of logging in AWS. Indeed, we focus on a particular logging mechanism and class of logs only: AWS CloudTrail for



```
{
  "Statement": [{
    "Effect": "Allow",
    "Action":
      [ "s3:PutObject", "s3:ListBucket" ],
    "Resource": "arn:aws:s3:::myBucket"  }]  }
```

**Figure 4.7.** An example AWS policy. It has an ‘Allow’ or ‘Deny’ effect, and the actions and the resource to which the policy applies. The policy may be attached to a user or role.

so-called management events [95]. AWS has other logging capabilities; for example, it is possible to log so-called service and data events [95], and deploy not only CloudTrail, but also CloudWatch [96].

### 4.5.1 Background

AWS is a public cloud provider: a customer can pay for and get computing resources at various layers of the software stack. One of their services is serverless computing: a customer writes call-back functions which are associated with events that may happen, and deploys those functions using a service called AWS Lambda [97]. The customer does not need to explicitly allocate computing resources. Hello, Retail! is such a serverless application. We now overview AWS IAM, CloudTrail and Hello, Retail!

**AWS IAM** AWS IAM (Identity and Access Management) enables you to manage access to AWS services and resources securely [98]. It implements a version of role-based access control. Their version combines role-based and identity-based authorization. IAM users may be assigned to IAM roles, and IAM roles assigned to IAM policies. An IAM user can be assigned to multiple roles, and an IAM role can be associated with multiple policies. An IAM user can assume a role and is then able to exercise the role’s access privileges. In addition, in IAM, a user can be associated directly with a policy. Figure 4.7 shows an example of a policy. A policy may also be attached directly to a resource such as an “S3 bucket”: S3 is AWS’s storage service [99]. In IAM, there is also the notion of a service role. This is a special IAM role that is associated with services that are managed by AWS such as Elastic Compute Cloud

(EC2) [100] and AWS Lambda [97]. These managed services assume a service role in order to obtain permissions to make API calls to other AWS services.

**CloudTrail** AWS CloudTrail allows us to log, continuously monitor, and retain account activity related to actions across your AWS infrastructure [95]. The logs are in JSON format and are delivered to an S3 storage bucket. CloudTrail characterizes events into data, service and management events. Data events log read/write access to objects in S3 and databases in DynamoDB. Service events are created by AWS services, and are not necessarily triggered by the cloud application. Management events log all control plane operations associated with a AWS account. For example, accessing a table in AWS DynamoDB would be logged as a data event, while creating a new IAM role would be logged as a management event. All access control changes made via IAM are logged by CloudTrail. We use these logs as the source of evidence to perform forensic analysis. Figure 4.8 shows an example of a CloudTrail log.

**Hello, Retail!** There are three reasons that we have chosen Hello, Retail! [82] for our case-study. One is that the serverless computing design pattern where applications are hosted by a third-party service such as AWS Lambda [97] is considered to be at a cutting-edge, and Hello, Retail! has won an award for its serverless architecture [85]. Another reason is that Hello, Retail! has been written to run on AWS, which provides both a rich access control scheme within AWS IAM [98], and logging capability within AWS CloudTrail [95]. A third reason is that included with Hello, Retail! is a security policy configuration: users and roles, and associated authorizations. This offers us the opportunity to deploy our analytical approach from the prior sections.

Hello, Retail! is an event sourcing application that has been made available as open source. The front-end interface of Hello Retail is a website hosted on AWS CloudFront, a content delivery network. Its back-end consists of multiple AWS Lambda functions, each of which is a call-back function each of which triggers on an event. The scenario for which Hello, Retail! has been built is that of a merchant adding a product to their online store. When this happens, two things must occur. Firstly a photographer needs to take a photo of the product. Secondly, the customers should see the new product with the new photo in the product catalog. The application comprises several components each of which runs as a

---

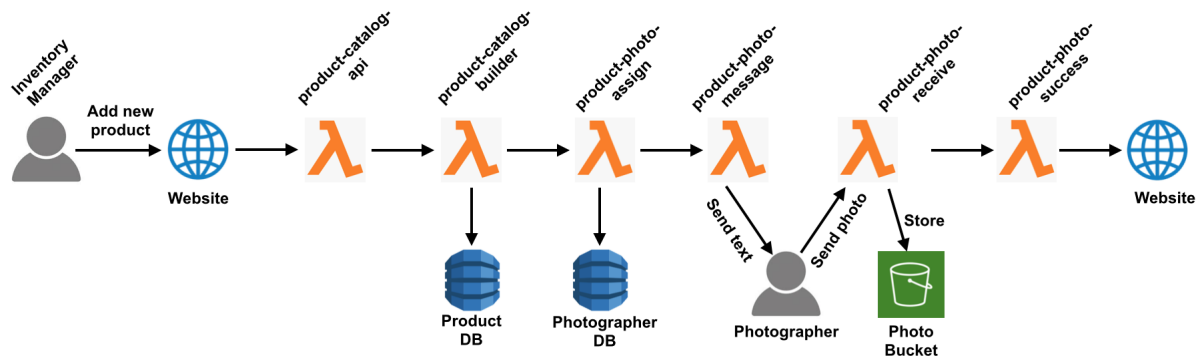
```

1  {"Records": [{
2    "eventVersion": "1.0",
3    "userIdentity": {
4      "type": "IAMUser",
5      "principalId": "EX_PRINCIPAL_ID",
6      "arn": "arn:aws:iam::123456789012:user/Alice",
7      "accountId": "123456789012",
8      "accessKeyId": "EXAMPLE_KEY_ID",
9      "userName": "Alice"
10   },
11   "eventTime": "20140324T21:11:59Z",
12   "eventSource": "iam.amazonaws.com",
13   "eventName": "CreateUser",
14   "awsRegion": "us-east-2",
15   "sourceIPAddress": "127.0.0.1",
16   "userAgent": "aws-cli/1.3.2 Python/2.7.5 Windows/7",
17   "requestParameters": {"userName": "Bob"},
18   "responseElements": {"user": {
19     "createDate": "Mar 24, 2014 9:11:59 PM",
20     "userName": "Bob",
21     "arn": "arn:aws:iam::123456789012:user/Bob",
22     "path": "/",
23     "userId": "EXAMPLEUSERID"
24   }}
25 ]}]

```

---

**Figure 4.8.** An example CloudTrail log. The log records a management event in which Alice creates a new user Bob in IAM.



**Figure 4.9.** Portion of Hello, Retail!'s workflow.

Lambda function. Depending on their purpose, some Lambda functions may require access to other AWS resources. For instance, the Lambda function whose purpose is to receive images from photographers requires access to AWS S3 [99] in order to store the image files in the relevant S3 photo bucket. A Lambda function obtains the required access it needs by assuming an IAM service role that has a policy attached to it that grants access to the relevant AWS resource.

Figure 4.9 illustrates the portion of Hello, Retail!'s workflow that pertains to us. An inventory manager logs onto the website and registers a new product. This event triggers the `product-catalog-api` function which in turn triggers the `product-catalog-builder` function, where the product's information is added into the product database. Next the `product-photo-assign` function is called which finds an available photographer from the photographer database and then calls the `product-photo-message` function. This latter function sends the photographer a text requesting for a picture of the product. `Product-photo-receive` waits to receive a text containing an image and when it does, it stores it into an S3 data bucket. The final function `product-photo-success` associates the image with the product in the product database and displays the image on the website.

## 4.5.2 Forensic Analysis

We now discuss an incident we effected in the Hello, Retail! application that is not expected, and forensic analysis to track down its cause. An ‘incident,’ in this context, is some anomaly that is observed, that is caused by a malicious act.

**Incident** For our incident, we did two things: (i) over-privileged a service role, and, (ii) caused an incorrect photo to be associated with a product. We explain (ii) first. We added two products, a MacBook and an iPhone, to the Hello, Retail! application. A photographer, Alice, is assigned to take a picture of the MacBook, while another photographer, Bob, is assigned to take a picture of the iPhone. Alice and Bob take the photos and send them back. However, on the front-end website, an iPhone is shown on the page of both products, while the expected behaviour is that a MacBook should be displayed on the MacBook’s product page and an iPhone should be displayed on the iPhone’s product page.

We now explain (i). The intent of (i) is to be the cause of (ii). That is, the reason for this incident occurring is that a malicious user altered the code of the `product-photo-receive` Lambda function (see Figure 4.9) so that it stored the incorrect image in the photo bucket. The root cause of our incident is (i), the over-privilege which allows an attacker to alter the code of the Lambda function. We do not say in what manner an attacker exploits this over-privilege of the service role. We can certainly extend our scenario to include this – for example, identify the service that assumes this service role, and then give the attacker sufficient privilege over that service in our application. We chose not to, to keep the incident and analysis somewhat limited for purposes of exposition. At the minimum, identification of this over-privilege immediately gives us a solution to address this security issue. Note, also, that over-privilege is recognized as one of the top 10 security problems for serverless technology [101], and was recently identified to be the root cause of the Capital One Hack that exposed the personal information of more than 100 million customers [102].

**Analysis** From the standpoint of analysis, there are a few possible reasons for the security incident occurring. The true cause, which is a malicious user’s modification of the code of a Lambda function, is one. As an example of another, the legitimate photographer may simply

have sent in the wrong image. Referring back to Figure 4.9, an algorithm to investigate the security incident would proceed as follows. It would first check if the photo that was received by the photographer was of a Macbook. If it was, then the photographer is exonerated and the logged behaviour of Lambda functions `product-photo-receive` and `product-photo-success` have to be checked to see which of them was modified. If the photo received from the photographer was of an iPhone, then before accusing the photographer, we must verify that the correct photographer was assigned by `product-photo-assign` and that the correct text message was sent to the photographer by `product-photo-message`. AWS CloudTrail logging for management events does not provide logs that are sufficient to run such a procedure. For instance, checking if the correct text message was sent to the photographer requires that all messages sent out to photographers by the `product-photo-message` function are logged; but this is not logged. The approach we discuss in Section 4.2 has promise, however.

That is, we investigate the security incident by considering the access control system and its state changes. We observe from Figure 4.9 that only `product-photo-receive` should have access to the S3 photobucket. However it is possible that a malicious user may have given another Lambda function access to S3 as well and then modified the function to change the photo assignment. To investigate which Lambda function had access to S3 resources in some past state, an investigator can issue a query for each of the Lambda functions. We wrote a program for this – see Algorithm 4 for its pseudocode. A query would take the form,  $\langle \gamma, \psi, k, q = \langle F, AmazonS3Access \rangle, \pi = Actual, L \rangle$ , where  $\gamma$  is the current state,  $\psi$  is the state change rule,  $k$  is the known past state of the IAM access model,  $q$  is the forensic query,  $F$  is the Lambda function,  $\pi$  is the access type and  $L$  is the CloudTrail logs. The program takes as input a query that comprises of a role  $R$  (e.g. `ProductPhotoReceiveRole`) and an access  $A$  (e.g. `S3ReadWriteAccess`), and the CloudTrail logs. The goal of the program is to determine from the logs whether in any past state the access  $A$  was assigned to the role  $R$ . It parses each log entry to determine what the event associated with it is. If the event is *AttachRolePolicy*, then two parameters of the event, *RoleName* and *PolicyArn* are checked to verify if they are identical to  $R$  and  $A$  respectively. If the condition is met, the program outputs the policy. If no log entry is found that allows the given access to the given role, the program outputs an empty set.

**Input:** role  $R$ , access  $A$ , set of log entries  $L$

**Output:**  $result \in \{True, False\}$

```
1  $S = \emptyset$ 
2  $result = \emptyset$ 
3 foreach  $l \in L$  do
4   if  $l.eventName = AttachRolePolicy$  then
5     if  $l.RoleName = R \wedge l.PolicyArn.Access = A$  then
        $result = result \cup l.PolicyArn$ 
6 return  $result$ 
```

**Algorithm 4:** Pseudocode to determine if a Lambda function's role had access to S3.

On running our program for each Lambda function, we found that, as expected, only the function `product-photo-receive` had access to S3. However, from the output of the program we found that the one of the policies giving the service role access to S3 was a *PowerUserAccess* policy. Such a policy grants read and write access to all S3 buckets, i.e., that service role is over-privileged. Hence we conclude that the `product-photo-receive` Lambda function's code may have been altered by a malicious user. From our forensic analysis, we can conclude also that the logging capability of AWS IAM allows for one to efficiently determine whether a principal actually possessed a privilege in the past.

**Goal-Directed Logging** We observed that even though CloudTrail logged all management events, only a small portion of the raw logs collected were actually related to our forensic queries. As the number of users in the system grows, and the number of events that are logged increases, the size of logs increases. This has two drawbacks. Firstly, the cost associated with storing the logs in S3 increases. Secondly, and more importantly, it leads to the *quantity problem* where the size of logs makes a meaningful analysis difficult [92, 93]. We therefore wanted to see how much the size of logs could be reduced by using goal-directed logging. As mentioned in Section 4.4, goal-directed logging is not only useful to decide in advance what needs to be logged, but also, in a situation in which one cannot predict what logs will be needed, it can be used to filter out logs before they are analyzed.

	Total # of Logs	Reduced # of Logs	% Reduction
Light Case	4491	47	98.95%
Heavier Case	23331	57	99.76%

**Table 4.1.** A summary of how the size of logs changes when goal-directed logging is used. As we discuss in the prose, the Heavier Case has five times the personnel as the Light Case.

We analyzed the benefits of goal-directed logging under two scenarios. For each scenario, we deployed the Hello, Retail! application to AWS, and added products and registered photographers to the application’s databases. We then simulated the security incident discussed above by changing `Product-Photo-Receive`’s code so that for the given item, a wrong photo is stored in the S3 photo bucket. In the first scenario, we simulated a light use case where there was only a single inventory manager and a single photographer registered in the system. In the second scenario, we simulated a heavier use case where there were 5 inventory managers and 5 photographers. We then compared the total number of logs generated by AWS, and the number of logs that remained after filtering using goal-directed logging.

Table 4.1 summarizes our results. In the first scenario, the total number of logs generated was 4491, out of which only 47 were related to our forensic queries of interest. That is, with the help of those 47 entries, we were able to generate the same forensic result as when using all 4491 entries. In the second scenario, the total number of logs generated was 23331 out of which only 57 were related to our forensic queries of interest. In both cases, goal-directed logging can reduce the number of logs by at least 98%. It is also interesting to observe that in going from the light use-case to the heavy use-case, the total number of logs entries increases by about a factor of 5 but the number of logs actually used by the forensic analysis increases by a factor of 1.2 only. From these numbers we can conclude that goal-directed logging has the potential to significantly reduce the size of logs that need to be analyzed for a forensic query.



## 4.6 Related Work

To our knowledge, forensics analysis has not been posed in access control, as such, in prior work as we do in this work. However, there has been work on computer forensics in more general contexts, and we summarize these below. We discuss also prior work on safety.

**Computer forensics** Early discussions about forensics in contexts to which our work pertains are in the work of Noblett et al. [103]. That work suggests that computer forensics dates back to around the mid-1990's. Peisert et al. provide five, high-level principles of forensic analysis [104]. That work proposes also a method to capture useful data rather than more data, using 'goal-oriented' logging in which logging requirements are extracted by considering the goals of an attacker [93, 105]. Several event reconstruction forensic tools have been proposed. Amongst the earliest is ReVirt which improves the integrity of logs by encapsulating the target system inside a virtual machine, and then placing the logging software beneath this virtual machine [59]. ReVirt replays the complete, instruction-by-instruction execution of the virtual machine to answer forensic queries. Zhu et al. proposed the Repairable File Service, which uses logs to identify files that potentially have been contaminated by an intrusive process [62]. King et al. introduced Backtracker, which helps an administrator backtrack from a file with suspicious contents to possible entry points of the intrusion using logged system calls [63]. Sitaraman et al. improved upon this backtracking method by logging additional parameters of the file system and using data flow analysis to prune unwanted paths [60]. Goel et al. proposed Forensix, which comprehensively logs kernel events, streams this information in real-time onto append-only storage and then uses database technology to support high-level querying of the archived log [61].

Peisert et al. proposed analyzing sequences of function calls for forensic analysis, focusing on isolating cause and effects of the intrusion attack, and looking not only for unexpected events but also for the absence of expected events [106]. Bishop et al. presented a technique for deriving auditing criteria, and thereby logging requirements, from security policies [94]. Several other projects such as CERT's Incident Detection, Analysis, and Response Project (IDAR) [107] and SRI's DERBI [108] assist administrators in understanding intrusions of file systems. The book by Carrier discusses file system forensics in depth [58]. Olivier transferred

concepts of file system forensics to database forensics [109].

Recent work has studied forensics in the context of the Internet of Things (see [110] for a survey), and cloud computing (see [111] for a survey).

**Safety Analysis** To our knowledge, safety was first posed by Harrison et al. [65] in the context of the access matrix model [112]. They introduced it as the question in which one asks if a certain right can ‘leak’ in the future and showed that in general, the problem is undecidable for the HRU scheme. However safety is decidable for the following restricted versions of HRU: (1) no subjects or objects are allowed to be created, (2) each command has at most one condition, and subjects and objects cannot be destroyed, and (3) each command comprises a single primitive operation. Tripunitara and Li [72] revisited the work of Harrison et al. [65] and introduced other notions of safety such as whether a particular right can leak into the cell associated with a particular object, or into the cell associated with a particular subject and object. In Section 4.3.1, we showed forensic analysis for HRU was undecidable by reducing from their safety notion which asks if a right can leak into the cell associated with a particular subject and object pair in which it did not exist in the current state.

Li et al. [67, 68] studied security analysis, a generalization of safety analysis, for trust management [77] — an approach to access control in decentralized distributed systems with access control decisions based on policy statements made by multiple principals. Security analysis generalizes simple safety analysis in which one asks if there exists a reachable state in which a specific principal has access to a specific resource. Li and Tripunitara [69] studied two classes of security analysis problems for RBAC. Both classes use variants of a component of the ARBAC97 administrative model for RBAC [79]. Jha et al. [81] studied the URA-RC-SAP security analysis problem (SAP) in RBAC with the URA97 administrative scheme [79]. The URA-RC-SAP problem allows assignments, revocations and trusted users. They study the computational complexity of safety analysis for various sub-schemes and establish that while the safety analysis is **PSPACE**-complete for the general scheme and some sub-schemes, for some other sub-schemes, it is **NP**-complete, and even in **P**. Solworth and Sloan introduced a new discretionary access control (DAC) scheme based on labels and relabelling that has decidable safety properties [73]. Li and Tripunitara [70] argue that DAC should not be equated

with the HRU access matrix scheme [65] in which safety is undecidable, and present an efficient algorithm for deciding safety in the Graham-Denning DAC scheme [80].

## **4.7 Conclusions**

Our conclusions are discussed in Chapter 6, Section 6.2.

# Chapter 5

## Logic Locking: Observations from Foundations and a Security Notion<sup>5</sup>

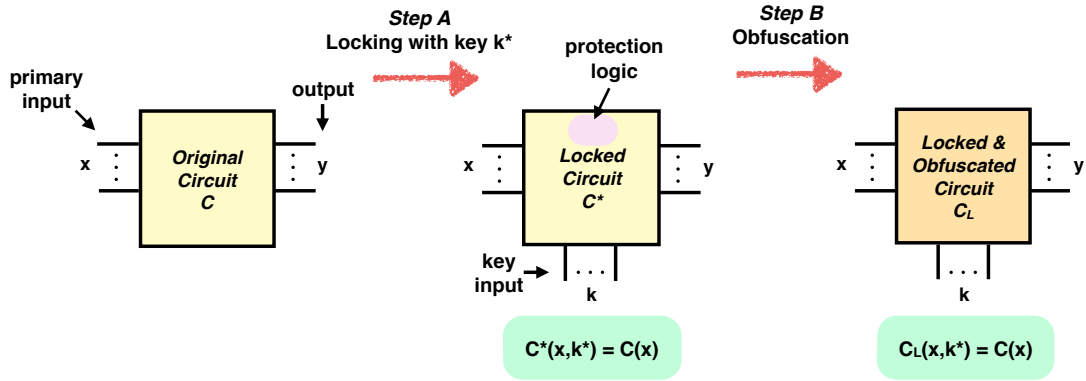
### 5.1 Introduction

We address *logic locking*, an approach that has been proposed to protect the Intellectual Property (IP) of the designer of a digital Integrated Circuit (IC, or simply, circuit) from a potentially untrustworthy semiconductor foundry [113]. Outsourcing of IC fabrication to such a foundry has economic advantages to the designer. However, it comes with a potential for the theft of their IP which is encoded as part of the design. If the design is provided to the foundry in the clear, the foundry could, for example, fabricate unauthorized copies of the IC.

Logic locking works as follows; see Figure 5.1. The IP designer selects a locking algorithm. The input to this algorithm is the circuit to be protected,  $C$ , and the designer's choice for the

---

<sup>5</sup>This chapter is based on portions of work that is undergoing review [4]. I was involved in designing a circuit that is resilient against the SAT attack when XOR-locked with a single key-bit (Section 5.2.1), explaining why XOR-locked benchmark circuits are not resilient against the SAT attack (Section 5.2.2), making observations on TTLock's security (Section 5.3.1), and explaining the conditions under which the attack on TTLock in Section 5.3.2 succeeds. The security notion presented in Section 5.4 is one that I formulated, and is different from the one presented in our paper [4].



**Figure 5.1.** Overview of logic locking. In Step A, the circuit to be protected is locked with a chosen key  $k^*$  using a locking algorithm. The algorithm inserts some protection logic to ensure that when the correct key is applied at the key inputs, the locked circuit is equivalent to the protected circuit. To prevent identification and removal of the protection logic, Step B obfuscates the output of Step A.

length of a *key*, which is a sequence of bits. Alternately, a key may be chosen and provided as input to the locking algorithm. The locking algorithm inserts some protection logic into the circuit, obfuscates this logic and then outputs a circuit  $C_L$  and a *correct key*  $k^*$ . The circuit  $C_L$  can be perceived as a two-input function,  $C_L(\cdot, \cdot)$ , where the first input is an input to the original circuit  $C$ , and the second input is a key. The circuit  $C_L$  has the property that  $C_L(x, k) = C(x)$  for all inputs  $x$  if and only if  $k$  is a correct key. We use the article “a,” rather than “the” for “correct key” because by intent of the logic locking algorithm or unintended by it, more than one key may be correct. The IP designer sends  $C_L$  to the foundry for fabrication. When the foundry sends back the fabricated chips, the IP designer loads a correct key onto tamper-proof memory on each chip.

We can contrast logic locking with other approaches that seek to achieve the same security objective. For example, following is a mechanism based on a universal circuit — a circuit that can simulate any circuit of a maximum size, given its description as input. Suppose the universal circuit is realized as a Field-Programmable Gate Array (FPGA). The designer encrypts the circuit to be protected using a semantically secure encryption scheme. The designer then programs the FPGA with the decryption logic given this encrypted circuit and a key. For a working IC, a correct key is provided. With such an approach, the only component whose fabrication can be outsourced is the FPGA. We do not consider such an approach logic

locking. To summarize, logic locking seeks to (1) outsource fabrication of the custom IC, perhaps with some added security logic, and, (2) seeks the overhead of the security features to be significantly lower than, for example, a universal circuit construction.

**Attacks** An *attack* on a logic locking mechanism has, broadly, one of two different, but related, objectives: (i) *key recovery*, and, (ii) *circuit recovery*. If an attack targets key recovery, then an attacker seeks to identify a correct key. If an attack targets circuit recovery, then an attacker seeks to output a circuit  $C'$  with the property that for all inputs  $i$ ,  $C'(i) = C(i)$ . A manner in which the two objectives are related is that the recovery of an entire correct key implies successful recovery of the circuit. However, the converse may not be true.

Either attack (i) or (ii) may be *approximate* [114]. For example, in the key-recovery attack, if the key-length is 64 bits, we may deem an attacker to have succeeded if they are able to correctly identify 60 bits of a correct key. Similarly, in the circuit-recovery attack, we may deem an attacker to be successful if she is able to recover a circuit whose output differs from that of the protected circuit for a small number of inputs only.

What the attacker is provided in attacks that have been considered in prior work and this work are (i) the locked and obfuscated circuit  $C_L$ , and, (ii) *blackbox access* to the original circuit  $C$ . “Blackbox access” means that the attacker is allowed to exercise  $C$  with some input  $i$  and get the output  $C(i)$  while incurring a cost of  $\Theta(1)$ . This can be considered realistic because the attacker may be able to purchase a working (unlocked) IC to use while attacking a locked copy of the IC. In some attacks, the attacker is weaker – they do not have (ii). This is typical of *removal attacks* that work by removing components of  $C_L$  (see, for example, our attack in Section 5.3.2).

**Contributions** We make two sets of contributions. The first is new observations, rooted in foundations and bolstered by implementations and empirical results, about two prior locking techniques and attacks. For each of the first set of observations, we conclude with “lessons learned.” These lessons, and lessons from prior work about locking techniques and security properties that have been proposed in prior work, inform our second contribution which is a new security property for logic locking.

In Section 5.2, we revisit a particular locking technique called XOR-Locking [113] with regards to its resistance to the SAT attack [115]. XOR-Locking is one of the earliest such schemes to have been proposed, to our knowledge, and the SAT attack, which targets key recovery, has received considerable attention in prior work. It is now customary, when one proposes a new logic locking scheme, to demonstrate, or even prove, its resilience to the SAT attack. “SAT,” in this context, is an acronym for the boolean satisfiability problem. The attack is based on the observation that a single input/output pair from a working IC can potentially enable an attacker to eliminate numerous incorrect keys.

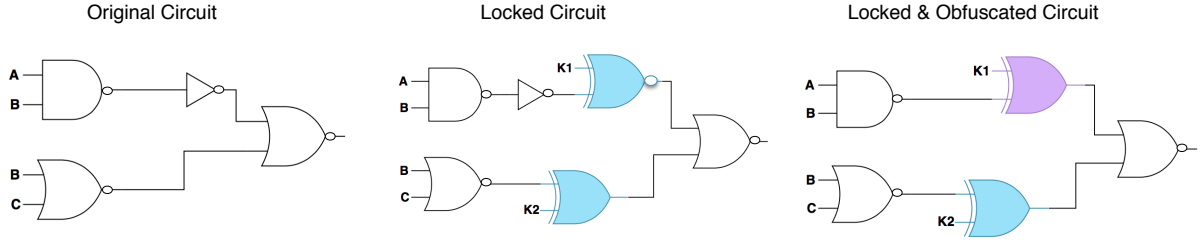
The SAT attack has been demonstrated, in practice, to successfully recover entire keys of XOR-Locked benchmark circuits [115]. Thus, it is appropriate to say that XOR-Locking is susceptible to the SAT attack. We ask what this means in more depth. Specifically, is it the case that given a circuit  $C$ , its XOR-Locked version  $C_L$  is susceptible to the SAT attack? We provide the answer: no, if one-way functions exist. A one-way function is a function  $f$  that is polynomial-time computable, and for a uniform choice  $x$ , given only  $f(x)$ , it is computationally hard to find  $x'$  such that  $f(x') = f(x)$  [116]. In other words, there exist circuits for which XOR-Locking is effective against the SAT attack, provided one-way functions exist. Indeed, we establish that this is the case even with a one-bit key. We are able to generate such circuits in practice under the assumption that one round of SHA-256 [117] is a one-way function, and have run the SAT attack against them and verified their invulnerability. We have made three such circuits available publicly [118]. Given the observation that such circuits exist, we then ask: is there something about the benchmark circuits and the manner in which they are XOR-Locked that makes them susceptible to the SAT attack? We answer this question by first focusing on the first step of the SAT attack, which, given a locked circuit, is the identification of whether every key is correct for it. We identify a property, which we can call wire redundancy (see Definition 4 in Section 5.2.2), of the benchmark circuits that if preserved by the locking and obfuscation mechanisms of XOR-Locking, is related to the problem of determining whether every key is correct for the corresponding XOR-Locked circuit. We do this using average-case computational complexity. Specifically, we identify that if checking for this property, i.e. wire redundancy, in an XOR-Locked circuit is in **distP**, then the problem of determining whether every key is correct for its XOR-Locked version is

in **distP**, where **distP** is the average-case analogue of the complexity class **P** (see Definition 8 in Section 5.2.2). We present empirical evidence that the benchmark circuits are indeed from a distribution for which checking for wire redundancy is computationally ‘easy’.

*An attack on TTLock* In Section 5.3, we address a state-of-the-art locking mechanism, TTLock [119], that has been claimed to be provably secure against the SAT attack. We first observe that the proof of security of TTLock relies on the assumption that a SAT solver chooses its output uniformly from the set of all possible satisfying assignments. We then point out, based on a plot from the work on TTLock [119], that this assumption is not supported by empirical evidence. We then present a circuit recovery attack against TTLock and show its effectiveness against benchmark circuits. Our attack is probabilistic, but not approximate: when we succeed, we recover the circuit in its entirety. We discuss the conditions under which our attack succeeds with significant probability. We have implemented the attack, and we present empirical results from attacks on benchmark circuits. We are aware that there is prior work that attacks TTLock [120, 121]; however, those are key-recovery attacks. Our attack recovers the circuit without having to recover the key. At the minimum, our attack identifies more vulnerabilities in TTLock than known previously.

*A Security Notion* The back and forth between defences and attacks in the logic locking space has been rightfully attributed to a lack of appropriate security notions [122]. In the absence of such notions, a newly proposed defence is customarily shown to be secure by demonstrating its resilience against known attacks. Our attack against TTLock, which is a scheme previously shown to be resilient against the SAT attack, serves as an example of why such an approach at “proving” security is faulty. That is, resilience against current attacks does not imply that a scheme is secure against future attacks. In Section 5.4 we propose a new notion of “secure” for logic locking. Our notion seeks to protect against a malicious foundry whose objective is to overproduce chips and sell them illegally in a black market. We compare our notion to security notions suggested by prior work [123, 4, 122].



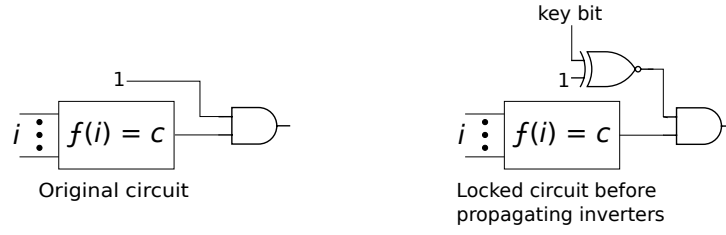


**Figure 5.2.** An example of XOR-Locking [113]. We begin with the original circuit  $C$  (left figure). To lock the circuit with a 2-bit key, we choose 2 wires, and insert XOR/XNOR key gates in to them (middle figure). The correct key bits are  $K1 = 1$  and  $K2 = 0$ . To prevent an attacker from knowing the correct key by simply observing the key gates, the inverter is propagated forward using the Boolean identity  $\neg x \oplus y = x \oplus y$  (right figure).

## 5.2 XOR-Locking

In this section, we address XOR-Locking [113]. Figure 5.2 illustrates how XOR-Locking works. We begin with an original, plaintext circuit  $C$ . We first identify wires in  $C$  into which we intend to insert two-input gates called key gates. Then, we insert either an XNOR or an XOR gate into each wire that was identified in Step (1). This yields a circuit which we denote as  $C_K$ . An XNOR gate corresponds to a key-bit of 1, and an XOR gate corresponds to a key-bit of 0. Finally, as an obfuscation mechanism, we propagate inverters forwards and/or backwards in  $C_K$  to yield the locked circuit,  $C_L$ .

It has been observed that XOR-Locking is ineffective against the SAT attack [115]. In the remainder of this section, we investigate this observation in more depth. In Section 5.2.1, we ask: do there exist circuits for which XOR-Locking is indeed effective against the SAT attack? Our answer is yes, under customary assumptions, with even a 1-bit key. Then, in Section 5.2.2, we identify, using average-case complexity, a sufficient condition for a circuit whose XOR-Locked version is susceptible to the first step of the SAT attack. We observe, empirically, that benchmark circuits do indeed satisfy this sufficient condition, which helps explain, from foundations, why XOR-Locking has been deemed to be susceptible to the SAT attack. In each of those sections, we conclude with lessons learned. To our knowledge, ours is the first work to study the strength and weakness of XOR-Locking in such technical depth.



**Figure 5.3.** A circuit (left) whose XOR-Locked version (right) with even a one-bit key is resilient to the SAT attack.  $f$  is a one-way function, and “ $f(i) = c$ ” is a circuit that outputs 1 if, on input  $i$ , the output of  $f$  is a constant  $c$  that is in the range of  $f$ ; otherwise its output is 0.

### 5.2.1 The Effectiveness of XOR-Locking

In this section, we ask: notwithstanding the supposed susceptibility of XOR-Locking to the SAT attack [115], do there exist circuits for which XOR-Locking is resilient to the SAT attack? The following theorem states that under a particular customary assumption, the answer is “yes,” with even a 1-bit key. The customary assumption is that one-way functions exist. This observation about the strength of XOR-Locking suggests that XOR-Locking can be effective.

**Theorem 5.2.1.** *If one-way functions exist, then there exists a family of circuits such that for every member in the family, XOR-Locking is resilient to the SAT attack with even a 1-bit key.*

Our proof is by construction, and it is shown in Figure 5.3. Suppose  $f$  is a one-way function, and  $c$  is a constant that is in the range of  $f$ ; that is, there exists some  $x$  such that  $f(x) = c$ . Suppose also that we know of no  $y$  such that  $f(y) = c$ . We first construct a circuit, shown in the box “ $f(i) = c$ ” in Figure 5.3, whose input wires encode an input  $i$  to the function  $f$ . The circuit “ $f(i) = c$ ” compares  $f(i)$  to  $c$  for equality. If  $f(i) = c$ , then the circuit “ $f(i) = c$ ” outputs 1; otherwise, it outputs 0. As the figure shows, our circuit  $C$  comprises the circuit “ $f(i) = c$ ,” whose output is the input of an AND gate. The AND gate’s second input is a constant 1 bit. The output of the AND gate is the one-bit output of  $C$ . Thus,  $C$  has the property that its output is 1 on input  $i$  if and only if  $f(i) = c$ , i.e., if and only if we have found an inverse of  $c$ .

A locked version of such a  $C$  with a 1-bit key is shown to the right of Figure 5.3. We insert an

XNOR or XOR gate in to the constant 1-wire in  $C$ . If we insert an XNOR gate as shown in the figure, then the correct key is 1; if we insert an XOR gate, the correct key is 0. No propagation of inverters, i.e., obfuscation, is necessary to thwart the SAT attack.

**Why this is hard for the SAT attack** The objective of the SAT attack is key recovery. Thus, for the above  $C$  and  $C_L$  in which the key gate is an XNOR, the SAT attack is successful if and only if it is able to determine that the correct key bit is 1. We reprise the algorithm that is the SAT attack as Algorithm 5.

**Input:** (i) Locked circuit  $C_L$ , and, (ii) Blackbox access to the original circuit  $C$

- 1 Let  $S = \emptyset$
- 2 Let  $F$  be a boolean formula that is satisfiable if and only if there exists keys  $k_1$  and  $k_2$ , and input  $i$ , such that for all  $x \in S$ ,  $C_L(x, k_1) = C_L(x, k_2) = C(x)$ , and  $C_L(i, k_1) \neq C_L(i, k_2)$
- 3 **while**  $F$  is satisfiable **do**
- 4      $S = S \cup i$
- 5 Find a key  $k^*$  such that for all  $x \in S$ ,  $C_L(x, k^*) = C(x)$
- 6 **return**  $k^*$

**Algorithm 5:** The SAT Attack [115].

The input to Algorithm 5 comprises the assets the attacker possesses i.e. the locked circuit  $C_L$  and an oracle to the plaintext circuit  $C$ . The algorithm begins by creating a set  $S$  that is initially empty (Line 1). Next it creates a boolean formula  $F$  that is satisfiable if and only if there exists an input  $i$  and two keys  $k_1$  and  $k_2$  such that  $k_1$  and  $k_2$  when used to unlock  $C_L$  produce the correct output for all inputs in  $S$  but give different outputs from each other for the input  $i$  (Lines 2-3). It then adds  $i$  as a member to  $S$  (Line 4). When it cannot find any more inputs to add to  $S$ , it finds a key  $k^*$  that is correct for all inputs in  $S$  (Line 5). It outputs  $k^*$  which is guaranteed to be a correct key (Line 6).

In Algorithm 5, each check of the while condition in Line 3 is an invocation to a SAT solver, as is the determination of a key  $k^*$  in Line 5. The SAT attack works by progressively identifying more and more members of a *Distinguishing Set* of inputs,  $S$ , to  $C$ .

**Definition 2** (Distinguishing Set). *Given a logic-locked circuit  $C_L$  of a plaintext circuit  $C$ , a Distinguishing Set for it is a set of inputs  $S$  to  $C$  with the following property: for a particular key  $k$ , if  $C(i) = C_L(i, k)$  for all  $i \in S$ , then it is the case that  $k$  is a correct key.*

Once the SAT attack finds a Distinguishing Set, the while loop of Lines (3) and (4) terminates. Then, the SAT attack, in Line (5), finds a key that produces the correct output for all the members of this set. Although the set of all inputs is a Distinguishing Set because a correct key produces the correct output for all inputs, the SAT attack hopes to find a small Distinguishing Set. The efficiency of the SAT attack is dictated by: (i) the number of iterations of the while loop of Lines (3) and (4), i.e., the size of the distinguishing set  $S$  to which the attack converges, and, (ii) the hardness of each instance in the check in Line (3), and Line (5).

Now consider what the SAT attack attempts to do for our circuit in Figure 5.3. In the first step, it attempts to determine whether there exists an input  $i$  to the circuit such that  $C_L(i, 1) \neq C_L(i, 0)$ , i.e., an input  $i$  to  $C$  that distinguishes the one-bit key 1 from the key 0. If it is able to find such an  $i$ , the set  $\{i\}$  is a distinguishing set, and the SAT attack can then identify the correct one-bit key in Line (6). That is, for the locked circuit from Figure 5.3, there exists a Distinguishing Set of size one, and any input that the SAT attack identifies in its first step is exactly such an input. Determining whether such an  $i$  exists for our locked circuit from Figure 5.3 is equivalent to determining whether the empty set,  $\emptyset$ , is a distinguishing set. We call this problem out in the following definition. The empty set is distinguishing if and only if every key is correct.

**Definition 3** (NOTEMPTYSETDIST). *NOTEMPTYSETDIST is the following problem: given as input a logic-locked circuit  $C_L$  of a circuit  $C$ , is every distinguishing set for  $C_L$  of minimum size non-empty?*

For any locked circuit  $C_L$ , a *certificate* for a “yes” instance of NOTEMPTYSETDIST is exactly a triple  $\langle i, k_1, k_2 \rangle$  such that  $k_1 \neq k_2$  and  $C_L(i, k_1) \neq C_L(i, k_2)$ . In the SAT attack, the identification of such a certificate comes “for free” when NOTEMPTYDISTSET is solved as the first step. The reason is that NOTEMPTYDISTSET is in **NP**, i.e., a “yes” answer to an instance is associated with a certificate that is efficiently-sized and can be verified efficiently.

And it is typical for a solver that corresponds to the class **NP**, e.g., a SAT solver, to return a certificate along with a “yes” answer.

With regards to the locked circuit in Figure 5.3 then, if the SAT attack is able to “easily” perform key recovery for some  $C_L$ , then it is able to “easily” solve NOTEMPTYSETDIST for that  $C_L$ . This means that the SAT attack has “easily” identified an  $i$  such that  $f(i) = c$ . This is because of the following. Suppose the key gate is, as the figure depicts, an XNOR, and we apply as one-bit key the value 0. Now the output of the XNOR gate in Figure 5.3 is 0, and therefore the output of  $C_L(\cdot, 0) = 0$ . To distinguish this key from the key 1, we require an input  $i$  to the circuit that causes the output of the locked circuit to be 1, i.e., we require  $C_L(\cdot, 1) = 1$ . We know that when the key-bit is 1, the output of the XNOR gate is 1. Thus, when the key-bit is 1, the output of  $C_L(i, 1)$  is 1 if and only if the output of the “ $f(i) = c$ ” circuit is 1. The SAT attack will have “easily” inverted the one-way function  $f$  for the value  $c$  from its range, whose pre-image was chosen uniformly.

**Comparison to prior approaches** Broadly, the idea of using a one-way function to thwart the SAT attack is not new. There is prior work that is similar to our construction in Figure 5.3, specifically work by Yasin et al. [124] and Zhou[125]. Their objective is to take an existing circuit  $C$ , and leverage constructs such as a one-way function to securely lock  $C$ . It is unclear that these prior schemes indeed achieve what they set out to do — their approaches do not provably guarantee that the SAT-attack will have to solve the one-way function in at least one of its iterations. In contrast, we seek to show only that there exist circuits for which XOR-Locking thwarts the SAT attack, and we are able to do this provably.

**Realization in practice** We have designed and implemented software that outputs instances of the plaintext and locked circuits as depicted in Figure 5.3, and confirmed that the implementation of the SAT attack of [115] indeed fails to solve NOTEMPTYSETDIST. “Fails,” in this context, means does not determine that NOTEMPTYSETDIST is indeed true within 12 hours on a desktop computer with an 8-core AMD Ryzen 7 2700 processor and 32 GB RAM that runs Ubuntu Linux 18.04. The function  $f$  we have adopted is one round of SHA-256 [117]. We generate the image of  $f, c$ , by first picking a random input that guarantees that only one round of SHA-256 is exercised, and then computing its image  $c$  in software using the

`java.security` package. We have made three instances of the original circuit, each of which corresponds to a different  $c$ , and a corresponding locked version available publicly [118]. Our circuit implementation of SHA-256 is based on [126].

**Lessons learned** The locked circuit from Figure 5.3 should be easy for an attack to break and it may be somewhat surprising that the SAT attack, which has been the focus of so much prior work on logic locking on account its effectiveness, fails for it. An effective attack on that locked circuit could observe, for example, that a key bit of 0 results in the output of the XNOR gate to be 0, which in turn causes the output of the AND gate, and therefore the circuit, to always be 0. Therefore, one can confidently output the correct key bit as 1.

However, one cannot fault an attack, in this case the SAT attack, as being too weak, particularly given that it has been demonstrated to be successful against benchmark circuits. Rather, we argue that the property that underlies the SAT attack is too weak. That is, a meaningful security property for logic locking would deem that the locking mechanism in Figure 5.3 is not necessarily secure.

## 5.2.2 The Ineffectiveness of XOR-Locking

The observation in the previous section, that there indeed exist circuits for which XOR-Locking mitigates the SAT attack, appears to go against the observation in prior work that XOR-Locked circuits are susceptible to the SAT attack. In this section, we provide a reasoning, based in computing foundations, about why this is the case for the benchmark circuits. That is, we ask: is there a property of the benchmark circuits and/or the XOR-Locking procedure, that renders XOR-Locked circuits from the benchmark easy for the SAT attack? The reason, in this question, that we identify the original plaintext circuit as a possible culprit in the success of the SAT attack is that our construction in the previous section exactly relies on a property of the original plaintext circuit to render the SAT attack ineffective.

The answer to the above question is ‘yes.’ The computing foundations to which we appeal is *average-case complexity* [6]. In classical computational complexity, we consider a decision problem,  $\mathcal{L}$ , and ask what upper- and lower-bounds are for its worst-case computational

hardness. In average-case complexity, we consider a *distribution problem* which is a pair,  $\langle \mathcal{L}, \mathcal{D} \rangle$ . The component  $\mathcal{L}$  is a decision problem as in classical complexity. The component  $\mathcal{D}$  is a family of distributions,  $\mathcal{D} = \{D_n\}$ , where each  $D_n$  is a distribution for problem-size  $n$ . For example,  $\mathcal{L}$  may be 3-CNF-SAT – satisfiability of boolean formulas in propositional logic that are in conjunctive normal form with exactly three literals per clause [127]. The distribution  $D_n$  may be that we uniformly pick an instance of size  $n$  whose ratio of number of clauses to number of variables is some constant. It is known that 3-CNF-SAT is computationally hard for particular such families of distributions, and easy for some others [128].

Average-case complexity is sensitive to the choice of the input distribution. A reason that considering average-case complexity is meaningful in this context is that we know, from the previous section, that in the worst-case for the attacker, XOR-Locking can indeed withstand the SAT attack. What we establish in this section, however, is that the distribution from which the benchmark circuits are drawn render their XOR-Locked circuits susceptible to the SAT attack. More specifically, what we do is the following. We identify a property of circuits which we call *wire redundancy*. We observe that it is a sufficient condition to make an inference about NOTDISEMPTYSET (see Definition 3 in the previous section). We then establish a correspondence between the computational ease of checking wire redundancy, and that of NOTDISEMPTYSET, using average-case complexity. We then observe empirically that the benchmark instances are indeed drawn from such a distribution.

**The notion of wire redundancy** Our first step in identifying a property that renders NOTDISEMPTYSET easy for the benchmark instances is to define what we call a redundant wire in a circuit. We do so in the following definition.

**Definition 4** (Redundant wire). *Given a circuit  $C$  and a wire  $w$  in  $C$ , we say that  $w$  is redundant if for every single-input, single-output circuit  $X$ ,  $C$  is equivalent to  $C^{(X)}$ , where  $C^{(X)}$  is exactly  $C$ , except with  $w$  replaced by  $X$ . By “equivalent,” we mean that  $C$  and  $C^{(X)}$ , for every input, produce the same output.*

**Theorem 5.2.2.** *A wire  $w$  in a circuit  $C$  is redundant if and only if  $C$  is equivalent to  $C^{(w)}$ , where  $C^{(w)}$  is exactly  $C$ , but with an inverter inserted in to the wire  $w$ .*

*Proof.* The “only if” direction is straightforward by contradiction. For the “if” direction,

assume otherwise for the purpose of contradiction. That is, assume that for some  $\langle C, w \rangle$ , it is the case that  $C$  is equivalent to  $C^{(w)}$ , and yet  $w$  is not redundant. As  $w$  is not redundant, there exists some circuit  $X$  and some input  $i$  such that  $C(i) \neq C^{(X)}(i)$ . It must be the case that the output of the sub-circuit  $X$  is  $\neg w$  on input  $i$  (i.e., the negation of whatever value  $w$  takes in  $C$  for input  $i$ ) and thus,  $C(i) \neq C^{(w)}(i)$ , a contradiction.  $\square$

Below, we articulate what it means to unlock an XOR-Locked circuit.

**Definition 5** (Unlocking an XOR-Locked circuit). *Let  $C_L$  be an XOR-Locked circuit of some circuit  $C$ . To unlock  $C_L$  with key  $k$ , we (1) hard code  $k$  as a key input for  $C_L$ , and (2) replace each key gate in  $C_L$  with either an inverter or a wire, in the obvious way, so  $C_L$ 's functionality is maintained.*

We now proceed to computational ease. We first address the XOR-Locked circuit. If it is easy to check, for each wire in an XOR-Locked circuit, whether the wire is redundant, then it is easy to solve NOTDISTEMPTYSET for the locked circuit. Towards this, we first define two more computational problems.

**Definition 6** (CHECKWIREREDUNDANCY). *CHECKWIREREDUNDANCY is the following problem: given as input a circuit,  $C$ , for every wire  $w$  in  $C$ , either find an input  $i$  such that  $C(i) \neq C^{(w)}(i)$ , or decide that  $w$  is redundant. The output on input  $C$  is a pair  $\langle i, w \rangle$  for each wire  $w$  in  $C$ , where  $i$  is either an input to  $C$  such that  $C(i) \neq C^{(w)}(i)$  or  $\phi$  if  $w$  is redundant.*

**Definition 7** (F-NOTEMPTYDISTSET). *F-NOTEMPTYDISTSET is the following problem: given as input an XOR-Locked circuit  $C_L$  of a circuit  $C$ , find an input  $i$  such that there exist two keys  $k_1, k_2$  for which  $C_L(i, k_1) \neq C_L(i, k_2)$ , or decide that  $\emptyset$  is a distinguishing set for  $C_L$ . The output on input  $C_L$  is either  $i$ , or a special symbol, say,  $\phi$ .*

Our intent with the above two definitions is to relate computational ease of checking wire redundancy with checking whether the empty set is distinguishing. A nuance with the above two definitions is that algorithms for them must output a solution, rather than only “yes” or “no.” This is the reason that we refer to them as computational problems, rather than decision problems. We relate the computational ease of the above two problems in Theorem 5.2.3 below.



But before we do so, in the following definition, we make precise what “computationally easy” means for a computational problem such as one of the two above, whose only instances we care about are drawn from a particular distribution. We associate the pair of problem and distribution with a complexity class called **distP** that defines what it means for a deterministic algorithm  $A$  to solve a distributional problem  $\langle \mathcal{L}, \mathcal{D} \rangle$  in polynomial time on average.

**Definition 8 (distP, adapted from Arora and Barak [6]).** Let  $\mathcal{D}$  be a distribution over  $\{0, 1\}^*$ ; we denote sampling from a distribution using a left arrow,  $\leftarrow$ . If  $A$  is an algorithm, let  $\text{time}_A(x)$  denote the running-time of algorithm  $A$  on input  $x$ . Let  $E[\cdot]$  denote expectation, and  $|s|$  denote the size of the encoding of a string  $s$ .

Given a computational problem  $\mathcal{L}$  and a distribution  $\mathcal{D}$ , we say that  $\langle \mathcal{L}, \mathcal{D} \rangle \in \mathbf{distP}$  if there exists an algorithm  $A$  for  $\mathcal{L}$  and positive constants  $c, \epsilon$  such that:

$$E_{x \leftarrow \mathcal{D}} \left[ \frac{\text{time}_A(x)^\epsilon}{|x|} \right] \leq c$$

Our definition above is exactly that of Arora and Barak [6], except that we specify it for problems whose solutions need to be certificates if the answer is “yes,” rather than “yes” or “no” only. The complexity class **distP** is the average-case analogue of the conventional class **P**. The former captures the notion of ease for the ‘average’ instance of  $\mathcal{L}$  drawn from some distribution  $\mathcal{D}$ , while the latter captures the notion of ease for the computational problem  $\mathcal{L}$  for the worst-case instance. We now relate the computational ease of checking wire redundancy with the ease of checking whether the empty set is distinguishing.

**Theorem 5.2.3.** Let  $\mathcal{D}$  be any distribution of XOR-Locked circuits. Then  $\langle \text{CHECKWIREREDUNDANCY}, \mathcal{D} \rangle \in \mathbf{distP}$  implies  $\langle \text{F-NOTDISTEMPTYSET}, \mathcal{D} \rangle \in \mathbf{distP}$ .

*Proof.* Let  $A$  be an algorithm that, in Definition 8, renders  $\langle \text{CHECKWIREREDUNDANCY}, \mathcal{D} \rangle$  to be in **distP**. We devise an algorithm  $A'$  that renders  $\langle \text{F-NOTDISTEMPTYSET}, \mathcal{D} \rangle$  to be in **distP**. The algorithm is as follows: on input locked circuit  $C_L$ , invoke  $A$  on  $C_L$ : if  $A$  has in its output set, for any wire  $j$  of  $C_L$  that corresponds to a key input, an input  $\langle i, k \rangle$  such that  $C_L(i, k) \neq C_L^{(j)}(i, k)$ , output  $i$ . Otherwise, decide that  $\emptyset$  is a distinguishing set for  $C_L$ .

Algorithm  $A'$  solves F-NOTDISTEMPTYSET. To see this, observe that if  $A$  produces as output an input  $\langle i, k \rangle$  for the wire  $j$  corresponding to the  $j$ th key bit, then it must be the case that  $C_L(i, k) \neq C_L(i, k^{(j)})$ , where  $k^{(j)}$  is exactly  $k$  but with the  $j$ th bit flipped. In the same vein, if  $A$  decides that no input  $\langle i, k \rangle$  as above exists for any of  $C_L$ 's key inputs, then it must be the case that for any input  $i$ , and any keys  $k_1$  and  $k_2$  that differ only on one bit,  $C_L(i, k_1) = C_L(i, k_2)$ . But it then follows that  $C_L(i, k_1) = C_L(i, k_2)$  for all  $i, k_1$  and  $k_2$ , because otherwise, not all of the wires that correspond to input key bits on which  $k_1$  and  $k_2$  differ can be redundant; at least one of those wires must be non-redundant. And if indeed  $C_L(i, k_1) = C_L(i, k_2)$  for all  $i, k_1, k_2$ , then all keys are correct and  $\emptyset$  is a distinguishing set for  $C_L$ .

It remains to be shown that the expectation of the running-time of  $A'$  is indeed bounded as expressed in Definition 8. Towards this, let  $c$  and  $\epsilon$  now be positive constants such that:

$$E_{C_L \leftarrow \mathcal{D}} \left[ \frac{\text{time}_A(C_L)^\epsilon}{|C_L|} \right] \leq c.$$

We show that a positive constant  $c'$  and a positive constant  $\epsilon'$  exist such that:

$$E_{C_L \leftarrow \mathcal{D}} \left[ \frac{\text{time}_{A'}(C_L)^{\epsilon'}}{|C_L|} \right] \leq c'.$$

To see this, observe first that  $A'$  requires linear time to search  $A$ 's answer for an input-key pair  $\langle i, k \rangle$  corresponding to one of  $C_L$ 's key inputs. In other words, for every  $C_L$  there is a constant  $c_1$  such that

$$\text{time}_{A'}(C_L) \leq c_1 |C_L| + \text{time}_A(C_L)$$

Now if we let  $\delta = \min(1, \epsilon)$  we have

$$\begin{aligned} \frac{\text{time}_{A'}(C_L)^\delta}{|C_L|} &\leq \frac{[c_1 |C_L| + \text{time}_A(C_L)]^\delta}{|C_L|} \\ &\leq \frac{[2 \max(c_1 |C_L|, \text{time}_A(C_L))]^\delta}{|C_L|} \\ &\leq \frac{[2c_1 |C_L|]^\delta}{|C_L|} + \frac{[2 \text{time}_A(C_L)]^\delta}{|C_L|} \\ &\leq 2c_1 + 2 \left[ \frac{\text{time}_A(C_L)^\epsilon}{|C_L|} \right]. \end{aligned}$$

It then follows that

$$\begin{aligned} E_{C_L \leftarrow \mathcal{D}} \left[ \frac{\text{time}_{A'}(C_L)^\delta}{|C_L|} \right] &\leq 2c_1 + 2 E_{C_L \leftarrow \mathcal{D}} \left[ \frac{\text{time}_A(C_L)^\epsilon}{|C_L|} \right] \\ &\leq 2c_1 + 2c \end{aligned}$$

□

**Relationship to the plaintext circuit,  $C$**  The above discussion that culminates in Theorem 5.2.3 relates a property of the locked circuit,  $C_L$ , with the first step of the SAT attack. We now ask and answer: what, if anything, does this have to do with an original, plaintext circuit,  $C$ , from the benchmarks on which the SAT attack has been demonstrated to be successful? We show that there is a property of the original benchmark circuit  $C$  that renders F-NOTDISTEMPTYSET easy provided the process of XOR-Locking causes the property to be preserved in  $C_L$ . The property is simply that checking wire redundancy is easy for the benchmark circuit itself.

**Theorem 5.2.4.** *If an XOR-Locking procedure does not propagate any inverters as in the right part of Figure 5.2, then for any circuit  $C$ , if an input  $i$  exists such that  $C(i) \neq C^{(w)}(i)$  for some wire  $w$  in  $C$ , then for any XOR-Locked version  $C_L$  of  $C$ , there exists an input  $i'$  such that  $C_L(i') \neq C_L^{(w')}(i')$ , where  $w'$  is the wire in  $C_L$  corresponding to wire  $w$  in  $C$ .*

*Proof.* Suppose there exists an input  $i$  such that  $C(i) \neq C^{(w)}(i)$  for some wire  $w$  in  $C$ . Now consider  $C_L$  with inputs  $i$  and  $k^*$ , where  $k^*$  is the correct key. Clearly  $C_L(i, k^*) \neq C_L^{(w)}(i, k^*)$ . □

In other words, if we can easily find a proof that a wire  $w$  in  $C$  is not redundant, then we have a proof that the corresponding wire in  $C_L$  is not redundant. So checking wire redundancy in a locked circuit is no harder than checking wire redundancy in the plaintext circuit.

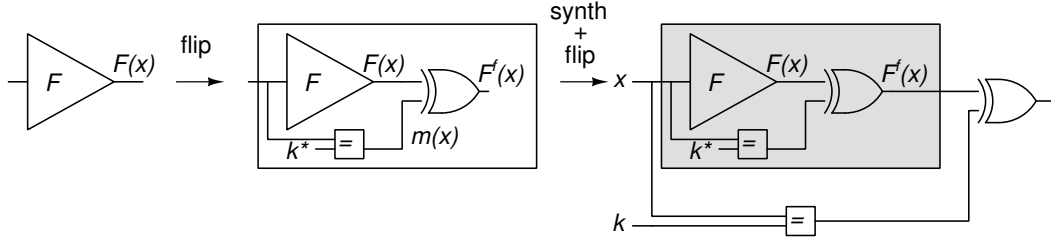
We have established, empirically, that the set of circuits in the benchmark does indeed meet the sufficient condition of Theorem 5.2.4. That is, CHECKWIREREDUNDANCY is indeed

“easy” for every circuit  $C$  in the benchmark. To observe this, we did the following. For every circuit in the benchmark, we check the redundancy of each wire in the circuit using ABC’s combinational equivalence engine of the ABC package. Our results are shown in the Table 5.1 below.

Benchmark	#In	#Out	#Gates	Time(s)	Benchmark	#In	#Out	#Gates	Time(s)
c432	36	37	160	6	c499	41	32	202	9
des	256	245	6473	465	c5315	178	123	2307	24
apex2	39	3	610	26	c7552	207	108	3512	211
apex4	10	19	5360	362	c880	60	26	383	16
c1355	41	32	546	24	dalv	75	16	2298	124
c1908	33	25	880	38	c2670	233	140	1193	58
c3540	50	22	1669	85	ex1010	10	10	5066	338
ex5	8	63	1055	48	i4	192	6	338	14
i7	199	67	1315	63	i8	133	81	2464	127
i9	88	63	1035	85	k2	46	45	1815	338

**Table 5.1.** Time it takes for our wire redundancy checking procedure to finish executing on the set of benchmark circuits from prior work [115].

**Lessons learned** We can immediately infer, based on our observations in this section, that the original circuit in Figure 5.3 whose 1-bit XOR-Locked version thwarts the SAT attack has at least one wire for which it is computationally hard to determine whether it is redundant. Also, this section and the previous one clearly point to properties of the original circuit as sources of hardness for the SAT attack. This in itself is not a bad thing; for example, an encryption scheme is not expected to mask the distribution from which a plaintext is chosen to be encrypted. Thus, based on our observations in this section and the previous, we cannot categorically say whether, for particular circuits of interest, XOR-Locking will be effective or not. Rather, what is needed is a clearly expressed security property against which we can check whether a particular mechanism such as XOR-Locking meets it.



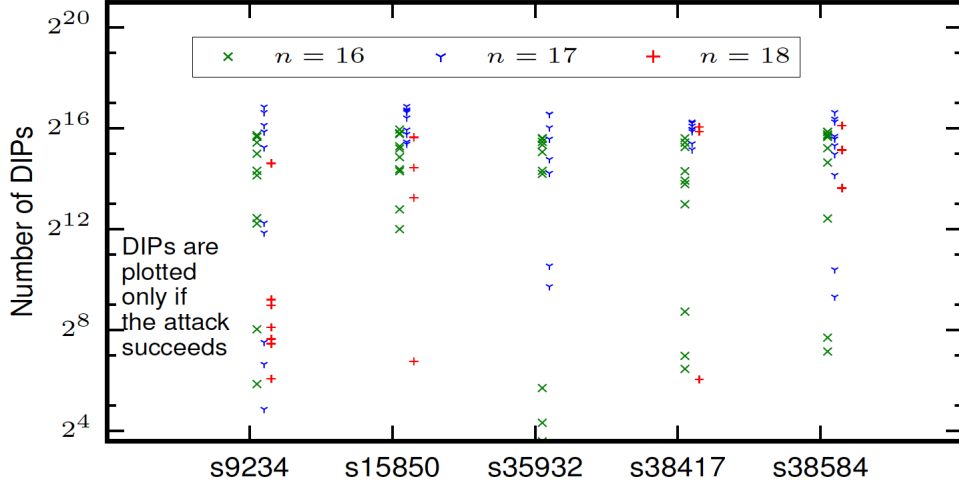
**Figure 5.4.** Example illustrating the TTLock (or SFLL-HD<sup>0</sup>) scheme [123].  $F$  is the function implemented by the circuit to be protected. First, protection logic is inserted to flip the output when the input is equal to the correct key (corresponding to Step A of Figure 5.1), then synthesis is performed (corresponding to Step B of Figure 5.1), and some more logic is inserted for the correct key to recover the original circuit (corresponding to Step A of Figure 5.1).

### 5.3 TTLock

In this section, we consider a current state-of-art defense, TTLock [119] and its subsequent generalization SFLL-HD<sup>m</sup> [123]. TTLock, pictured in Figure 5.4, works as follows. Let  $C$  be the circuit to be protected and let  $F$  be the function implemented by  $C$ . First, TTLock selects a secret key  $k^*$  and creates a circuit that implements  $F^f$ , where  $F^f(x) = F(x)$  for all  $x \neq k^*$ , and  $F^f(k^*) = \neg F(k^*)$ .  $F^f$  is created by adding a comparator and an XOR gate to the original circuit.

Then, the designer synthesizes the circuit implementing  $F^f$  using a synthesis tool, which can be thought of as an algorithm that restructures a Boolean circuit to reduce its area. From the standpoint of TTLock, this synthesis is intended to obfuscate the circuit for  $F^f$ , specifically, mix the comparator, which has the secret key  $k^*$  hard-coded as input, with  $F$ , so that an adversary is not able to identify them in a locked circuit. However, the particular synthesis procedure one should adopt is not specified as part of TTLock, and therefore, it is unclear that this choice to adopt synthesis as obfuscation meets its intent.

Finally, a key input  $k$  is introduced, as well as another comparator and XOR gate. The comparator checks if  $k = x$ , and if yes, the XOR gate flips the output of  $F^f$ . This is the final



**Figure 5.5.** Figure, reprised from [119], showing the number of DIPs required for the SAT attack against TTLock for key-size  $n = 16, 17, 18$ , across 10 trials, for each of five benchmark circuits.

locked circuit  $C_L$ . Setting  $k = k^*$  recovers a circuit that is equivalent to the original circuit  $C$ , but setting it to any other value results in a circuit whose output differs from that of  $C$  on exactly two inputs:  $x = k^*$  and  $x = k$ .

SFLL-HD $^m$  generalizes TTLock by flipping  $F$  if the hamming distance between input  $x$  and  $k^*$  is exactly  $m$ . As such, TTLock is simply SFLL-HD $^0$ . The claimed benefit of  $m > 0$  is that the functions corresponding to incorrect keys are more different than the designer’s function, but in practice, setting  $m > 4$  makes the scheme susceptible to the SAT attack [123].

### 5.3.1 Observations on TTLock’s Security

TTLock is claimed to be resilient against the SAT attack because each new distinguishing input (DIP) found by the attack (assuming it is not equal to  $k^*$ ) eliminates one key only. For a key of size  $n$ , the expected number of DIPs needed for the SAT attack to break TTLock is  $2^{n-1}$  [119]. This is based on the assumption that given a satisfiable Boolean formula, a SAT solver will choose an assignment uniformly at random from the set of all possible satisfying assignments [119].

It is however unlikely that this assumption is true. Indeed, if it was, then the number of DIPs required would depend only on  $n$ , the size of the key, and not on the original circuit  $F$ . This is of course a desirable property but does not seem to be supported by empirical data. We reproduce in Figure 5.5 a plot from the work of Yasin et al. [119] for the number of DIPs required by the SAT attack for five ISCAS’85 benchmarks across 10 trials for each benchmark. Our first observation from the plot is the divergent nature of the distribution of the numbers for each of the benchmarks for  $n = 18$ . Our second observation are the points in the plot which fall well below the expected value for the number of DIPs. If the SAT solver chooses uniformly amongst satisfying assignments, then the probability that the number of DIPs required would be below 256 for  $n = 16$  across 10 trials is less than 4%. Yet we observe that for at least 4 out of the 5 benchmarks, 2 out of the 10 trials (i.e. 20%) had DIPs below 256. These observations suggest that the SAT solver is in fact biased towards the correct key. We therefore call the purported security of TTLock against the SAT attack into question.

Yet another observation that casts further doubt on TTLock’s security is that assuming that an attacker has access to the plaintext circuit,  $C$ , finding a key for its TTLocked version,  $C_L$ , is in **NP**. This is because given a key  $k^*$  as a certificate, one could verify that it is a correct key by checking that for  $x^* = k^*$ ,  $C_L(x^*, k^*) = C(x^*)$ , because only the correct key produces an output that is correct for an input that is equal to it. A consequence is that the problem of recovering a key for a TTLocked circuit can be reduced to SAT and be solved by a *single* call to a SAT solver. The query to the SAT solver asks if there exists an assignment to  $k$  and  $x$  such that  $(C_L(k, x) = C(x)) \wedge (x = k)$ . Consider a more general version of TTLock that has the following properties: (1) there exists a special input for which all keys except the correct key produce the wrong output, and (2) every input  $x$  that is not the special input eliminates one incorrect key only, but this incorrect key may not be equal to  $x$ . Suppose the attacker has only blackbox access to the plaintext circuit  $C$  and that SAT solvers were programmed to be able to use plug-in oracles. In this case also, an attacker would be able to find the correct key with a single call to the SAT solver. The oracle to the plaintext circuit,  $C_O(x)$ , is ‘plugged-into’ the SAT solver, and the solver is asked if there exists an assignment to  $k_1, k_2, k$ , and  $x$  that satisfies the following formula:

$$(C_L(k_1, x) \neq C_O(x)) \wedge (C_L(k_2, x) \neq C_O(x)) \wedge (C_L(k, x) = C_O(x)) \wedge (k_1 \neq k_2)$$

### 5.3.2 A New Attack on TTLock

We demonstrate a simple yet effective circuit-recovery attack against TTLock. We begin by noting that the security of TTLock is not premised on gates (comparator and XOR) that are outside the grey box in Figure 5.4. These are in the clear and can be easily found and removed.

Next, we hypothesize that the matching node  $m(x)$ , which is the output of the comparator that checks if  $x = k^*$  in the circuit that implements  $F^f$ , still remains in the circuit even *after* synthesis. That is, there exists a node in the synthesized circuit such that when the node is set to logic 0, the function  $F$  of the original circuit is recovered.

We then observe that the node  $m(x)$  has a special property — it is logic 1 for one and only one input. We define the signal probability of a logic gate in a circuit as the probability that for a randomly picked input, it will output 1. We say the signal probability of a gate is low if it is negligibly small. The signal probability of the gate whose output is  $m(x)$  is low — only  $\frac{1}{2^n}$  for an  $n$ -input circuit. Our attack uses this property of  $m(x)$  to identify it in the locked circuit.

Starting from a locked circuit  $C_L$  from which the external XOR and comparator have been removed, we proceed in three steps:

1. We estimate the signal probability for each gate in  $C_L$ . To do this, we apply  $N$  randomly selected inputs to  $C_L$ , and for each gate, we record the fraction of inputs for which it outputs a 1.
2. If the signal probability estimate for a gate is 0, i.e., the gate never activates for any of the  $N$  input patterns, we replace the gate with a constant 0.
3. We output the resulting circuit.

**Results** We tested our attack on circuits from standard benchmark suites, including the



MCNC and the ISCAS-89 benchmarks. For each circuit, we created one hundred different locked instances, selecting the key for each trial at random. We used the ABC package [129] which implements a commonly used open-source synthesis tool to synthesize the locked circuits. We applied  $N = 2000$  input patterns to estimate the signal probabilities of gates. Table 5.2 reports the percentage of trials for which we were able to recover an equivalent circuit for each of the benchmarks<sup>6</sup>. For all benchmarks except one, we recovered the designer’s function in at least 82% of the trials.

Benchmark	# Inputs	# Outputs	# Gates	% successes
scf	34	63	794	98
s420	34	17	151	82
s5378	214	228	2779	99
s13207	700	790	7951	67
s35932	1763	2048	16065	98
s38417	1664	1742	22179	94
s38584	1464	1730	19253	98
s15850	611	684	9772	88

**Table 5.2.** Number of trials out of 100 where our attack recovers the designer’s function for circuits from the MCNC and ISCAS’89 benchmarks.

**Why our attack works** We now investigate the conditions under which our attack is successful. First observe that if the original circuit  $C$  contains any gates whose output wires are redundant (see Definition 4 in Section 5.2), whether or not these wires activate for any of the  $N$  trials of the attack bears no consequence to the success of the attack. Therefore we can safely assume that  $C$  contains only gates whose output wires are non-redundant.

Next we observe that if the original circuit  $C$  does not contain any gates that have low signal probability, and if the synthesis tool does not modify the input circuit, then we can expect our attack to succeed with probability almost 1. This is because the output wire of each gate in  $C$  will activate for at least one of the  $N$  trials with probability almost 1.

---

<sup>6</sup>To check the equivalence of each reconstructed circuit with its corresponding benchmark, we used ABC’s combinational equivalence engine.

To check whether the benchmark circuits did not have any low signal probability gates, we repeated our attack on them without passing the locked circuits through the synthesis tool. We found that the attack succeeded for 6 out of the 8 benchmarks. That is, 6 out of the 8 benchmarks did not contain low signal probability gates. The 2 benchmarks that contained low signal probability gates, and therefore for which the attack, when launched on unsynthesized locked benchmarks, failed, were s13207 and s38584.

Our experiment, although limited, suggests that real-life circuits tend not to contain low signal probability gates. This is not surprising considering that a designer typically tries to optimize the utility of logic, and hence if the output of a gate is seldom active, the designer may avoid using the gate altogether. This also suggests a natural trade-off between security and efficiency in a circuit — if some redundancy is allowed in a circuit, it may be possible to obtain more security, at least from our attack.

The theorem below gives a lower-bound on the probability that our attack would be successful on a TTLocked circuit, if the synthesis algorithm does not modify the input circuit in any way.

**Theorem 5.3.1.** *If the synthesis algorithm is the identity function i.e. it does nothing to the input, then for a circuit  $C$  with  $n$  inputs and  $g$  gates, the probability that our attack succeeds on TTLocked  $C$ , has a lower-bound of:*

$$\left[1 - g(1 - s)^N\right] \left(1 - \frac{1}{2^n}\right)^N,$$

where  $s$  is the minimum signal probability of a gate in  $C$ .

*Proof.* If the synthesis algorithm is the identity function, then the attack succeeds if and only if two conditions hold: (1) The output wire of each gate in  $C$  is logic 1 for at least one of the  $N$  trials and, (2) The output wire  $m(x)$  is logic 0 for all the  $N$  trials. If  $s$  is the minimum signal probability of a gate in  $C$ , then the maximum probability that an output wire of a gate in  $C$  will not activate during a trial is  $1 - s$ . The probability that the output wire of a gate will not activate for any of the  $N$  trials is  $(1 - s)^N$ . Using the union bound, the probability that at least one of the  $g$  gates will not activate for any of the  $N$  trials is at most  $g(1 - s)^N$ . Hence the probability that each gate will activate for at least one of the  $N$  trials has a lower-bound of

$1 - g(1 - s)^N$ . Given that  $m(x)$  is logic 1 for one input only, the probability that it will be logic 0 for all  $N$  trials is  $(1 - \frac{1}{2^n})^N$ .  $\square$

The lower-bound in Theorem 5.3.1 depends on the following parameters: (1)  $g$ , the number of gates in  $C$ , (2)  $s$ , the minimum signal probability of a gate in  $C$ , (3)  $N$ , the number of trials, and (4)  $n$ , the number of input bits in  $C$ . If  $g$  increases, the probability that at least one of the  $g$  gates will not activate for any of the  $N$  trials increases, and hence the probability of the attack's success decreases. If  $s$  increases, it is more likely that a gate will activate in a trial and therefore the probability of the attack's success increases. If  $N$  increases, the probability of the attack's success may increase or decrease. Finally, if  $n$  increases, the number of inputs for which  $m(x)$  is low increases, and hence the probability of the attack's success increases.

We next consider the general case in which the synthesis algorithm may not be the identity function. The next theorem states sufficient conditions under which the success probability of the attack is guaranteed to be high. The notation  $D'$  denotes the circuit that is obtained when output wires of low signal probability gates in the circuit  $D$  are grounded, i.e., set to 0.

**Theorem 5.3.2.** *The success probability of our TTLock attack is guaranteed to be high if the following conditions hold:*

1. *The original circuit  $C$  contains no low signal probability gates.*
2. *The synthesis tool is such that  $C'_{in}$  is functionally equivalent to  $C'_{out}$ , where  $C_{in}$  and  $C_{out}$  are the input and the output circuits of the synthesis tool respectively.*

*Proof.* If the first condition holds, then the resulting circuit produced from the first step of TTLock contains only one low signal probability gate, i.e., the one whose output is  $m(x)$  (see Figure 5.4). We refer to this circuit as  $C_{in}$  as it is input to the synthesis tool.  $C'_{in}$  is equivalent to the original circuit  $C$  because grounding  $m(x)$  in  $C_{in}$  recovers  $C$ . If the Condition (2) in the statement of the theorem holds, then  $C'_{out}$ , which is the output of our TTLock attack, is equivalent to  $C'_{in}$ , which in turn implies that  $C'_{out}$  is equivalent to  $C$ .  $\square$

Condition (2) in the theorem implies that the synthesis tool does not introduce non-redundant low signal probability wires in its resulting circuit. In this case the synthesis is unable to destroy an unfortunate property of TTLock — identifying  $m(x)$  in the output of the first step of TTLock allows one to recover the protected circuit.

**Concurrent work** Concurrently to our work on this attack, Sirone et al. [120] demonstrated a different attack procedure, although based on the same intuition that node  $m(x)$  is not removed by the synthesis tool. Instead of using signal probability, that work checks for certain properties of the node; for example, positive unateness, i.e., the value of  $m(x)$  increases monotonically in  $x$ . Sirone et al. [120] demonstrate their attacks work even on general SFL- $\text{HD}^m$  schemes for varying  $m$ . We demonstrate our attack on SFL- $\text{HD}^0$ , which is TTLock.

Yang et al. [121] present an attack against SFL that exploits structural traces left behind in the locked circuit. They propose two different techniques to carry out their attack, one is based on Gaussian elimination and the other uses knowledge of one protected input pattern to recover the key.

Both the attacks by Sirone et al. [120] and Yang et al. [121] are key-recovery attacks, whereas our attack on TTLock is a circuit-recovery attack. Furthermore, both their attacks on TTLock recover the correct key using structural traces left behind by the point function<sup>7</sup> that is implemented by the comparator that checks if the input is equal to the correct key. Hence, in theory, both their attacks on TTLock can be protected against by using a point function obfuscator<sup>8</sup>[130] to obfuscate the comparator. Our attack does not attempt to find the correct key and hence cannot be protected against using point function obfuscation.

**Lessons learned** A SAT attack does not account for possible removal of circuit components, which our attack in this section does. Any notion of security of logic locking must account

---

<sup>7</sup>A point function with target  $i \in \{0, 1\}^*$  is a circuit that on input  $i' \in \{0, 1\}^{|i|}$  returns 1 if  $i' = i$  and 0 otherwise.

<sup>8</sup>A point function obfuscator takes as input a circuit  $C$  that implements a point function for some target  $i$  and returns a circuit  $C_{ob}$  that is functionally equivalent to  $C$  i.e.  $C_{ob}$  is a point function for the same target  $i$ . Security requires that  $C_{ob}$  hides  $i$ .

for such classes of attacks as well. It must account also for probabilistic, yet exact (and not approximate), attacks. Our attack is probabilistic; however, when it succeeds, the entirety of the circuit is recovered.

## 5.4 A Security Notion

As we discuss in Section 5.5, research in the area of logic locking has mostly been a ‘cat and mouse’ game — when a defence is proposed, it is not long before an attack against it is discovered. The reason for this has been rightfully attributed to a lack of appropriate security notions for logic locking [122]. In the absence of such notions, locking schemes have been shown to be ‘secure’ by demonstrating their resilience against known attacks. However, as is suggested by our results from the previous two sections, the resilience of a locking algorithm against a specific attack or a class of attacks does not imply that it will not be vulnerable to future attacks. In this section we attempt to formulate a suitable security notion for logic locking. We do so by considering the threat model — the assets that the attacker possesses and the goals they want to achieve.

**Threat Model** A realistic threat model is one in which the attacker (i.e. the foundry) has the following assets: (1) the netlist of the locked circuit, (2) blackbox access to an unlocked circuit, and (3) knowledge of the locking mechanism that was used. (1) is true because the designer sends the foundry the netlist of the locked circuit for fabrication. (2) is true because assuming that the IC has already gone through at least one round of production, the foundry can purchase a working chip (an unlocked circuit) from the market. (3) is similar to Kerckhoff’s principle for cryptographic systems which says that such a system should be secure even if everything about the system, except the key, is public knowledge [131].

Possible goals of a malicious foundry are: (A) to sell the designer’s intellectual property (IP) to a competitor, and (B) to over produce chips and sell them illegally in a black market. In this work we differentiate between these two goals, and we attempt to devise a security notion for when the attacker’s goal is (B) only. Observe that an adversary that achieves goal (A) does not necessarily achieve goal (B). For example, consider when the adversary is only able to

recover a fraction of the original circuit. Competitors may be interested in purchasing this incomplete circuit but the circuit may not match the original circuit's output for sufficiently many (if any) inputs and hence it may not be possible to use it to produce illegal chips. On the other hand, it is safe to assume that an adversary that achieves goal (B) can achieve goal (A) too. This is because to achieve goal (B), the adversary must recover a circuit that matches the original circuit for all except possibly a small number of inputs. Such a circuit would be of interest to the IP designer's competitors. A scenario in which an IP designer may want to protect against (B) only is when they are a monopoly and have no competitors, or when an association overlooks malicious activities by competitors and holds them accountable for their actions thereby successfully deterring them from engaging in IP theft.

The circuit that is to be protected,  $C$ , can be thought of as an efficient circuit description of the function that is implemented by the corresponding working chip. The attacker's goal is to find a polynomial-sized circuit,  $C_{rec}$ , that is equivalent to  $C$ . Suppose the attacker finds a candidate  $C_{rec}$  and sells it illegally in a black market. The buyer may have access to a legal working chip purchased from the IP designer. For the buyer to verify that  $C_{rec}$  is indeed equivalent to a circuit description for the authentic working chip, they would have to compare the output of  $C_{rec}$  with that of the working chip for all inputs, which is not feasible. Therefore, it is reasonable to assume that an error margin can be tolerated. In other words, even if  $C_{rec}$  differs from  $C$  in its output for a set of inputs, if the cardinality of this set is sufficiently small, it may still be possible for a malicious foundry to use  $C_{rec}$  successfully. An observation is that the illegal chips sold in the black market are used as a blackbox, i.e., the buyer does not obtain the netlist of  $C_{rec}$ , but rather just an oracle to it. To capture these ideas formally, we use the terms *negligible* and *distinguisher* that are used in the field of cryptography.

**Definition 9.** A function  $f$  from the natural numbers to the non-negative real numbers is *negligible* if for every positive polynomial  $p$  there is an  $N$  such that for all integers  $n > N$  it holds that  $f(n) < \frac{1}{p(n)}$  [132].

A *distinguisher* is an algorithm, possibly a probabilistic one, that attempts to differentiate between two kinds of things. For example, in the context of pseudorandom generators (PRGs), the distinguisher attempts to differentiate between the result of applying a PRG to a random string, and a bunch of random bits [132].

**Definition 10.** A distinguisher,  $D$ , in the context of logic locking, is a (possibly probabilistic) polynomial-time algorithm who is given access to an oracle. We use the notation  $D_{C_1, C_2}$  to indicate that the oracle is formed from the two circuits  $C_1$  and  $C_2$ , each having equal length inputs, say  $n$ . When  $D_{C_1, C_2}$  gives an  $n$ -bit query  $x$  to the oracle it receives as output a string. In Game 0, the string is  $C_1(x)|C_1(x)$  where ‘|’ is concatenation. In Game 1, the string is  $C_1(x)|C_2(x)$ .  $D$ ’s goal is to ascertain whether it is playing Game 0 or Game 1. If  $D$  decides it is playing Game 0, it outputs 0, else it outputs 1.

Let  $P(D_{C_1, C_2}^{Game0} \rightarrow 0)$  and  $P(D_{C_1, C_2}^{Game1} \rightarrow 0)$  be the probabilities that  $D_{C_1, C_2}$  outputs a 0 when it is playing Game 0 and Game 1 respectively. Let  $W_{D_{C_1, C_2}}$  be the difference in these probabilities, i.e.,  $W_{D_{C_1, C_2}} = |P(D_{C_1, C_2}^{Game0} \rightarrow 0) - P(D_{C_1, C_2}^{Game1} \rightarrow 0)|$ .  $W_{D_{C_1, C_2}}$  can be thought of as the distinguisher’s advantage in telling oracles to  $C_1$  and  $C_2$  apart.

We can define the malicious foundry’s goal (B) formally as follows:

**Definition 11.** Given a locked circuit,  $C_L$ , whose input is of size  $n$  bits, and oracle access to the corresponding protected circuit,  $C$ , the goal of the attacker,  $\mathcal{A}$ , is to output a polynomial-sized circuit,  $C_{rec}$ , such that for some distinguisher  $D$  of interest to the attacker,  $W_{D_{C, C_{rec}}} \leq \mu(n)$ , where  $\mu$  is a negligible function.

Let  $\mathcal{A}'$  denote an ‘ideal’ adversary that has blackbox access to the protected circuit  $C$  and knowledge of  $C$ ’s size only, i.e.,  $\mathcal{A}'$  does not have access to the locked circuit  $C_L$ . The following is a security notion for logic locking that is based on the attacker’s goal in Definition 11. In the notion,  $C_{rec}$  and  $C'_{rec}$  are circuits output by  $\mathcal{A}$  and  $\mathcal{A}'$  respectively.

**Definition 12.** Let Lock be a locking mechanism, that on input circuit  $C$  and key-length  $\lambda$  outputs a locked circuit  $C_L$  and a corresponding  $\lambda$ -bit key  $k$ . Let  $\mathcal{C} = \{C_n\}_{n \in \mathbb{N}}$  be a class of circuits where  $C_n$  is a set of circuits on inputs of size  $n$ . We say that Lock is secure for  $\mathcal{C}$  if the following is true: for every probabilistic polynomial-time (PPT) adversary  $\mathcal{A}$  and every distinguisher  $D$ , there exists an ideal adversary  $\mathcal{A}'$  such that if there exists a negligible function  $\mu(n)$  such that  $\forall n, \forall C \in C_n, W_{D_{C, C_{rec}}} \leq \mu(n)$ , then there exists a negligible function  $\mu'(n)$  such that  $\forall n, \forall C \in C_n, W_{D_{C, C'_{rec}}} \leq \mu'(n)$ .

Informally the above notion says the following. If Lock is a locking algorithm that is secure from the possibility of the foundry achieving goal (B), then if an adversary  $\mathcal{A}$  can output a reconstructed circuit  $C_{rec}$  such that a distinguisher that is given oracle access to  $C_{rec}$  and  $C$  cannot tell them apart, then it must be possible for  $\mathcal{A}$  to obtain  $C_{rec}$  without access to  $C_L$ .

**Comparison with Security Notions of Yasin et al. [123]** Yasin et al. provide notions of security that are based on existing attacks, namely the SAT attack [115], the sensitization attack [133] and the removal attack [134]. For each of these attacks, they provide a security notion that characterizes a locking algorithm's degree of resilience against the attack. The disadvantage with such an approach is that a single security notion cannot capture the overall security of a locking algorithm. In contrast, our notion seeks to characterize whether or not an algorithm is secure against all known and future attacks that can be used by a foundry.

The SAT attack and sensitization attack are key-recovery attacks. The removal attack is a circuit-recovery attack. If a logic locking algorithm is susceptible to any key-recovery attack such that a foundry can find a key that unlocks the locked circuit, the locking algorithm is not secure by Definition 12. Similarly, if a logic locking algorithm is susceptible to a removal attack such that an equivalent of the protected circuit can be found, the locking algorithm is not secure by Definition 12. Therefore our security notion in Definition 12 is at least as strong as the notions proposed by Yasin et al.

There exists a locking algorithm that is not susceptible to the SAT-attack or to the sensitization attack, but is not secure under the notion given in Definition 12. An example is TTLock [119] which was shown to be resilient against both the SAT attack and the sensitization attack [123]. TTLock is not secure under the notion in Definition 12 because it is susceptible to other attacks that recover the protected circuit from the locked circuit, as shown in Section 5.3.2. Hence our notion in Definition 12 is stronger than the notions of Yasin et al. that are based on the SAT and sensitization attacks.

**Comparison with Security Notions of El Massad et al. [4]** Concurrently with this work, El Massad et al. proposed a new locking mechanism and proved that it possesses a security property that is akin to ciphertext indistinguishability in cryptography. Ciphertext indistinguishability means that no adversary, given an encryption of a message randomly



chosen from a two-element message space determined by the adversary, can identify the message choice with probability significantly better than that of random guessing. In a similar vein, the locking mechanism proposed by El Massad et al. which employs a sub-exponential indistinguishability obfuscator [135] and a puncturable pseudorandom function [136], guarantees that a computationally bounded adversary is unable to distinguish between any two circuits of the same size, regardless of the functions they implement, once they are locked. Their security property ensures that *any* information an adversary learns from a locked circuit and blackbox access to the protected circuit, can also be learned by an adversary that has knowledge of the size of the protected circuit and blackbox access to it only. Therefore their security notion protects against both possible goals, (A) and (B), of the attacker, and is stronger than the notion in Definition 12. Thus, their locking mechanism is an example of a scheme that is secure under Definition 12.

One may argue that a locking algorithm that forbids an adversary from learning anything from the locked circuit is superior than one that is secure under Definition 12 only. While this is certainly true from a security standpoint, it may not be true from other standpoints of interest such as the circuit area overhead. Therefore if an IP designer is interested in protecting against goal (B) only, it may be advantageous for them to consider a locking algorithm that is secure under our weaker notion in Definition 12. For example, consider a hypothetical locking algorithm whose protection logic has a small circuit footprint, and for which we can guarantee that any attack can recover at most a fraction of the protected circuit and this particular fraction of the circuit cannot be used to determine the output of any input correctly. This algorithm would be sufficient to protect against a malicious foundry whose goal is to sell illegal chips in the black market. Such an algorithm is secure under the notion in Definition 12 but is insecure under the notion of El Massad et al. [4]. A concrete example of a locking mechanism that is secure under Definition 12 but not under the notion of El Massad et al. [4] is an open problem.

**Comparison with Security Notions of Shamsi et al. [122]** Concurrently with this work, Shamsi et al. [122] proposed multiple security notions for logic locking. In their Exact Functional Secrecy (EFS) notion, the attacker’s goal is to find an *exact* circuit — one that is equivalent to the protected circuit for all inputs. In their Approximate Functional Secrecy notion, the attacker’s goal is to find an *approximate* circuit — one that is equivalent to the

protected circuit for all except at most a fraction of inputs. Their mindset is that a locking algorithm is secure under these notions if the success probability of an attacker who has access to the locked circuit and can make a polynomial number of calls to an oracle for the plaintext circuit is not significantly different from that of an attacker who does not have access to the locked circuit, can make a polynomial number of queries to the oracle, and randomly guesses the outputs that it does not obtain from the oracle.

They show via long-standing results in computational learning theory why it may be impossible to satisfy the approximation-resiliency notion of security for low depth combinational circuits under oracle-guided attacks. This is because many classes of Boolean functions are learnable from random or chosen input-output pairs in polynomial time at a rate significantly better than learning their truth-table. Hence, instead of bothering with attacking the locked circuit, the attacker can directly try to learn the protected circuit from the oracle to it with significant statistical advantage.

To avoid the negative results, they introduce the relaxed notion of Best-Possible Approximate Functional Secrecy in which the best the attacker can do is to try to black-box-learn the plaintext circuit  $C$  through input-output queries with the knowledge that  $C$ 's size is bounded to a known value. In other words, the locked circuit, beyond what the attacker can extract from the oracle through querying, reveals nothing new except a bound on the size of  $C$ .

Unfortunately, the way in which their security notions are formalized makes a direct comparison with our security notion difficult.

## 5.5 Related Work

There has been considerable work on logic locking over the past decade. To our knowledge, logic locking was first proposed in 2008 by Roy et al. [113]. In their scheme, EPIC, a set of XOR/XNOR key gates is randomly inserted into the original circuit, such that one input of each key gate corresponds to a key-bit and the other input is driven by an existing wire in the original circuit. Applying the correct key to the key-bits makes the locked circuit

equivalent to the original circuit. Rajendran et al. [133] observed that an adversary with access to a functional chip could attack the EPIC scheme by using automatic test pattern generation (ATPG) algorithms to reveal the key bits. They proposed a scheme based on key gates interference graph that is resilient against such attacks.

In 2015, El Massad et al. [137] and Subramanyan et al. [115] concurrently published attacks against Rajendran et al.'s scheme [133]. Their attacks are referred to in the literature as the SAT attack and it is based on the observation that a single input/output observation from a working chip can potentially eliminate a large number of incorrect keys. The discovery of the SAT attack was considered seminal because it defeated all locking schemes known at the time [138, 139, 140].

Much of the subsequent work focused on devising SAT-resilient schemes. These can broadly be categorized into three: (1) Those that attempt to formulate locking and obfuscation mechanisms that are not translatable to SAT, (2) Those that make the underlying SAT problems hard, and, (3) Those that attempt to make the number of SAT problems the SAT attack must solve in order to be successful exponential.

Examples of defences in the first category are cyclic obfuscation [141] and behavioral locking of the logic [142]. In cyclic obfuscation, the key idea is to introduce logical loops in the circuit so that it can no longer be represented as a directed acyclic graph. This forces a SAT attack to either be trapped in an infinite loop or to generate an incorrect key upon termination [143]. In behavioral locking, the key into a locked circuit not only determines the circuit's functionality, but also its timing profile. A functionality-correct but timing-incorrect key will result in timing violations. Shortly after the introduction of these schemes, stronger attacks such as cycSAT [144] and the Satisfiability Module Theories (SMT) attack [145] were discovered that were able to model cyclic or behavioral locking into a SAT or SAT+theory solvable logic problems.

Defences in the second category include Cross-lock [146] and Full-lock [147]. In Cross-lock, a onetime programmable interconnect mesh is used to obfuscate the routing of the circuit in order to substantially increase the runtime of each iteration of the SAT attack. In Full-lock a set of cascaded fully programmable logic and routing block (PLR) networks replace parts of

the logic and routing in the desired netlist, resulting in harder SAT instances than Cross-lock. Other defences in this category are those that incorporate one-way functions to make the underlying SAT instances hard [124, 125]. See Section 5.2.1 for a discussion on the last two schemes.

Defences in the third category include those that are based on point functions [148, 149, 150]. For example in SARLock [148], every incorrect key is wrong for the single input that is identical to it. Similarly, the Anti-SAT block scheme [149] utilizes functions whose number of input vectors that result in an output of 1 is either very high or very low. Although these defences make the number of iterations involved in the SAT attack exponential, they are vulnerable to other attacks such as the Signal Probability Skew (SPS) attack [134], removal attacks [151], approximate attacks [114, 152], the bypass attack [153], and the Satisfiability Module Theories (SMT) attack [145]. Also in the third category is the more recent and state-of-the-art defence TTLock [119] and its subsequent generalization SFLL [123]. Key-recovery attacks against TTLock and SFLL include the Functional Analysis Attack [120], and an attack that exploits structural traces left behind in the circuit [121]. We address TTLock in Section 5.3 and present a circuit-recovery attack against it.

Although some of the above mentioned schemes claim “provable” security, they only provide proofs against existing attacks, and do not formalize a precise notion of security as we do in this work. Yasin et al. [123] attempt to formulate definitions of security but their definitions are all predicated on existing attack strategies, specifically the SAT-attack [115], sensitization attack [133] and SPS removal attack [134]. Šišeković et al. [154] propose a unifying metric that attempts to capture the key security aspects of logic encryption. The goal of their work is to propose quantification metrics that can be used to compare different locking schemes. Other works that have proposed security notions are those by El Massad et al. [4] and Shamsi et al. [122], we discuss these in Section 5.4.

To the best of our knowledge, ours is the first work that: (1) Seeks to understand the strengths and weaknesses of XOR-locking from a foundational perspective, (2) Presents a circuit-recovery attack against TTLock that does not require the correct key to be found, and, (3) Proposes a security notion that differentiates between the attacker’s goals of sale of illegal

chips and sale of IP to a competitor, and seeks to protect against the former but not the latter.

## **5.6 Conclusions**

Our conclusions are discussed in Chapter 6, Section 6.3.

# Chapter 6

## Conclusions

In this chapter we summarize the insights that we have gained from appealing to computing foundations for each of the three problems studied in this thesis. We also discuss open problems that remain.

### 6.1 On VM-Scheduling to Combat Cloud Side-channels

In Chapter 3, we have addressed problems related to the use of scheduling as an approach to mitigating cross-VM side-channel attacks in cloud systems. Such attacks can cause information to leak. Specifically, we have sought to analyze the overhead incurred by an algorithm that seeks to minimize information leakage, for models of information leakage that have been proposed in prior work [5]. We have posed and answered a fundamental question: is the problem of determining a VM placement that minimizes information leakage tractable? We have established that it is intractable (**NP**-hard) by showing that even seemingly simpler sub-cases such as placing VMs across only two equal-capacity servers, and migrating VMs across only two equal-capacity servers, is **NP**-hard. Given the customary assumption,  $\mathbf{P} \neq \mathbf{NP}$ , this tells us that no efficient algorithm exists for the problem. We have identified, however, that a decision version to which the optimization problem relates to polynomially is in **NP**.

We have then analyzed prior work, which proposes an efficient, greedy approach called Nomad, motivated by an observation there that an approach based on reduction to ILP does not scale to even modestly-sized inputs that can arise in practice. We have identified that the reduction to ILP there can be inefficient, i.e., exponential-time. Also, we have identified deficiencies with Nomad: its initial design is unnecessarily inefficient, and the final design yields no decrease in information leakage for infinitely many inputs, unless  $\mathbf{P} = \mathbf{NP}$ , even though for these instances a reduction in leakage is possible.

We have then designed and implemented efficient reductions to ILP and CNF-SAT, with the intent of adopting corresponding solvers as oracles. We have conducted a limited empirical assessment with a focus on the main points of our work. Our empirical results suggest that the overhead imposed by an approach that indeed decreases information leakage is more than prior work suggests, but that the news is not all bad. For inputs that prior work suggests take longer than a day with an ILP approach, our implementation that is based on an efficient reduction takes a few minutes only. This suggests that further research-investment in well-founded approaches can decrease the overhead further, for instances that are likely to arise in practice.

Apart from such a research-investment as future work, we have pointed to several open problems. Is the closed migration problem tractable under models other than  $\langle R, C \rangle$ ? Is the general problem only weakly  $\mathbf{NP}$ -hard? Is the optimization version of the problem efficiently approximable?

## 6.2 On Forensic Analysis for Access Control

In Chapter 4, we have motivated, posed and studied forensic analysis in the context of access control systems, and observed its similarities to and differences from the more traditional safety analysis, of which forensic analysis seems a natural complement. We have instantiated forensic analysis for three access control schemes from the literature and identified that the worst-case computational complexity of forensic analysis for each of those schemes is the same as safety analysis. We have investigated also sufficient and minimal logs to render analysis for those schemes efficient (polynomial-time) and in that context, motivated and

discussed goal-directed logging. We have discussed a case-study of an open source serverless cloud computing application in which we have instantiated our analytical ideas as validation in practice.

There is tremendous scope for future work, both at the foundations and in practice. At the foundations, there is the issue of generalizing forensic analysis similar to the manner that safety analysis has been generalized to so-called security analysis to consider more general properties. In each case, we need to carefully consider meaningful ways to model these in the context of forensic analysis. In every such context, we need to keep in mind the additional, important issue of logging that arises with forensic analysis. From the standpoint of practice, we need to address the several different settings in which it is meaningful to pose forensic analysis in the manner we do, and further investigate the utility of goal-directed logging. We also need to consider realistic attacks that are anticipated in the future, and ask whether forensic analysis in the manner we pose and address is useful to mitigate those.

## **6.3 On Logic Locking to Protect Digital ICs**

In Chapter 5, we have made two sets of contributions on logic locking. The first is deeper observations on existing schemes and attacks on them. We have considered XOR-Locking [113], a scheme that has been shown to be vulnerable to the SAT attack [115]. We have asked whether there exists circuits for which XOR-Locking can be effective against the SAT attack, and have shown the answer to be ‘yes’ provided one-way functions exist. We have given examples of such circuits that are built from SHA256 and have empirically demonstrated that they are resilient against the SAT attack even when XOR-Locked with a single bit. We then asked if there exists a property of plaintext circuits that makes their XOR-Locked versions vulnerable to the SAT attack. We have found that the ease of checking wire redundancy in plaintext circuits is related to the ease of the problem that is solved by the first iteration of the SAT attack. The second scheme we have considered is TTLock [119]. We have made important observations on TTLock’s security and outlined a new removal attack that we have found is effective against TTLocked benchmark circuits. We have analyzed the



conditions under which our attack is successful. Our second contribution is a new security notion that seeks to protect against a foundry whose goal is over-production only. We have compared our notion to those proposed by prior work.

The promise of logic locking is certainly enticing as a technically sound scheme can revolutionize security of digital ICs. However, much work remains to be done. An open question pertains to a recent challenge that was published, and that is: should we consider an attack model in which the attacker has access to not only the locked circuit,  $C_L$ , but also the original circuit  $C$ ? In such an attack model, the goal of the attacker is key-recovery. We observe that with benchmark circuits as well, an attacker has complete knowledge of those circuits.

In this work we distinguished between two possible goals a malicious foundry may have: (A) to sell the designer's intellectual property (IP) to a competitor, and, (B) to over produce chips and sell them illegally in a black market. We provided a security notion that protects against (B) but not (A). An open question is whether it is possible to obtain security against goal (B) but not goal (A). The answer to this question would be 'yes' if there exists a construction of a locking scheme that is secure under our security notion but not under the stronger notions of prior work [4]. Whether such a scheme exists and, what, if any, benefit such a scheme offers over those that are secure under stronger notions, are open problems.

# Bibliography

- [1] Nahid Juma, Jonathan Shahren, Khalid Bijon, and Mahesh Tripunitara. “The Overhead from Combating Side-Channels in Cloud Systems using VM-Scheduling”. In: *IEEE Transactions on Dependable and Secure Computing* (2018). ©2020 IEEE.
- [2] *Avoid Infringement upon IEEE Copyright.*  
<https://journals.ieeeauthorcenter.ieee.org/choose-a-publishing-agreement/avoid-infringement-upon-ieee-copyright/>.  
Accessed: 2020-05-18.
- [3] Nahid Juma, Huang Xiaowei, and Mahesh Tripunitara. “Forensic Analysis for Access Control”. In: *Undergoing review*. (2020).
- [4] Mohamed El Massad, Nahid Juma, Jonathan Shahren, Mariana Raykova, Siddharth Garg, and Mahesh Tripunitara. “Logic Locking to Protect Digital Integrated Circuits (ICs): Observations from Foundations, and a Security Property and Mechanism”. In: *Undergoing review*. (2020).
- [5] Soo-Jin Moon, Vyas Sekar, and Michael K. Reiter. “Nomad: Mitigating Arbitrary Cloud Side Channels via Provider-Assisted Migration”. In: *Proceedings of the 2015 ACM Conference on Computer and Communications Security*. 2015, pp. 1595–1606.
- [6] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 2009. ISBN: 0521424267.
- [7] Walter J. Savitch. “Relationships between nondeterministic and deterministic tape complexities”. In: *Journal of Computer and System Sciences* 4.2 (1970), pp. 177–192.
- [8] Jonathan Shahren. “Using Polynomial Timed Reductions to Solve Computer Security Problems in Access Control, Ethereum Smart Contract, Cloud VM Scheduling, and Logic Locking.” PhD thesis. University of Waterloo, 2020.

- [9] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “A Shared Cache Attack That Works across Cores and Defies VM Sandboxing—and Its Application to AES”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2015, pp. 591–604.
- [10] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. “Wait a minute! A fast, Cross-VM attack on AES”. In: *International Workshop on Recent Advances in Intrusion Detection*. 2014, pp. 299–319.
- [11] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2015, pp. 605–622.
- [12] Rodney Owens and Weichao Wang. “Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines”. In: *30th IEEE International Performance Computing and Communications Conference*. 2011, pp. 1–8.
- [13] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. “Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds”. In: *Proceedings of the 2009 ACM Conference on Computer and Communications Security*. 2009, pp. 199–212.
- [14] Kuniyasu Suzuki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. “Memory Deduplication As a Threat to the Guest OS”. In: *Proceedings of the Fourth European Workshop on System Security*. 2011, 1:1–1:6.
- [15] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *23rd USENIX Security Symposium*. 2014, pp. 719–732.
- [16] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. “Cross-VM Side Channels and Their Use to Extract Private Keys”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 2012, pp. 305–316.
- [17] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. “Cross-Tenant Side-Channel Attacks in PaaS Clouds”. In: *Proceedings of the 2014 ACM Conference on Computer and Communications Security*. 2014, pp. 990–1003.
- [18] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. “Eliminating Fine Grained Timers in Xen”. In: *Proceedings of the 2011 ACM Workshop on Cloud Computing Security*. 2011, pp. 41–46.
- [19] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. “Resource Management for Isolation Enhanced Cloud Services”. In: *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. 2009, pp. 77–84.

- [20] Erman Pattuk, Murat Kantarcioglu, Zhiqiang Lin, and Huseyin Ulusoy. “Preventing Cryptographic Key Leakage in Cloud Virtual Machines”. In: *23rd USENIX Security Symposium*. Aug. 2014, pp. 703–718.
- [21] David Evans, Anh Nguyen-Tuong, and John Knight. “Effectiveness of Moving Target Defenses”. In: *Moving Target Defenses*. 2011, pp. 29–48.
- [22] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding applications from an untrusted cloud with haven”. In: *ACM Transactions on Computer Systems* 33.3 (2015), pp. 1–26.
- [23] Michael Sindelar, Ramesh K. Sitaraman, and Prashant Shenoy. “Sharing-aware algorithms for virtual machine colocation”. In: *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. 2011, pp. 367–378.
- [24] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979. ISBN: 0716710447.
- [25] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*, pp. 85–103.
- [26] Narendra Karmarkar and Richard M. Karp. “An Efficient Approximation Scheme for the One-dimensional Bin-packing Problem”. In: *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*. SFCS ’82. Washington, DC, USA, 1982, pp. 312–320.
- [27] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. 1st ed. New York, NY, USA, 2008. ISBN: 052188473X.
- [28] S. A. Vavasis. “Quadratic Programming is in NP”. In: *Information Processing Letters*, 36.2 (Oct. 1990), pp. 73–77.
- [29] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. 2nd. 2001. ISBN: 0070131511.
- [30] Carsten Sinz. “Towards an Optimal CNF Encoding of Boolean Cardinality Constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. 2005, pp. 827–831.
- [31] Nima Mousavi. “Algorithmic Problems in Access Control.” PhD thesis. University of Waterloo, 2014.
- [32] *CPLEX*. <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/>.
- [33] *Lingeling*. <http://fmv.jku.at/lingeling/>.

- [34] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. “Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud.” In: *IACR Cryptology ePrint Archive* (2015), p. 898.
- [35] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. “Attack directories, not caches: Side channel attacks in a non-inclusive world”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2019, pp. 888–904.
- [36] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache template attacks: Automating attacks on inclusive last-level caches”. In: *24th USENIX Security Symposium*. 2015, pp. 897–912.
- [37] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-channel attacks: Deterministic side channels for untrusted operating systems”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2015, pp. 640–656.
- [38] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. “Preventing your faults from telling your secrets”. In: *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS), Xian, China*. 2016.
- [39] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. “STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud”. In: *21st USENIX Security Symposium*. 2012, pp. 11–11.
- [40] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. “Limiting Cache-based Side-channel in Multi-tenant Cloud Using Dynamic Page Coloring”. In: *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*. 2011, pp. 194–199.
- [41] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. “Scheduler-based defenses against cross-vm side-channels”. In: *23rd USENIX Security Symposium*. 2014, pp. 687–702.
- [42] Yinqian Zhang and Michael K. Reiter. “Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud”. In: *Proceedings of the 2013 ACM Conference on Computer and Communications Security*. 2013, pp. 827–838.
- [43] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. *A Software Approach to Defeating Side Channels in Last-Level Caches*. arXiv preprint, arXiv:1603.05615v1. <http://arxiv.org/>. 2016.
- [44] Fangfei Liu and Ruby B. Lee. “Random Fill Cache Architecture”. In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014, pp. 203–215.

- [45] Zhenghong Wang and Ruby B. Lee. “A novel cache architecture with enhanced performance and security”. In: *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. 2008, pp. 83–93.
- [46] Dan Page. *Partitioned Cache Architecture as a Side-Channel Defence Mechanism*. 2005. URL: <http://eprint.iacr.org/2005/280>.
- [47] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. “CATalyst: Defeating last-level cache side channel attacks in cloud computing”. In: *2016 IEEE International Symposium on High Performance Computer Architecture*. 2016, pp. 406–418.
- [48] Oded Goldreich and Rafail Ostrovsky. “Software Protection and Simulation on Oblivious RAMs”. In: *Journal of the ACM* 43.3 (May 1996), pp. 431–473.
- [49] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. “Path ORAM: An Extremely Simple Oblivious RAM Protocol”. In: *Proceedings of the 2013 ACM Conference on Computer and Communications Security*. 2013, pp. 299–310.
- [50] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. “Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu”. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 2017, pp. 7–18.
- [51] Yulong Zhang, Min Li, Kun Bai, Meng Yu, and Wanyu Zang. “Incentive compatible moving target defense against vm-colocation attacks in clouds”. In: *IFIP International Information Security Conference*. 2012, pp. 388–399.
- [52] Yossi Azar, Seny Kamara, Ishai Menache, Mariana Raykova, and Bruce Shepard. “Co-Location-Resistant Clouds”. In: *Proceedings of the 2014 ACM Workshop on Cloud Computing Security*. 2014, pp. 9–20.
- [53] Yi Han, Tansu Alpcan, Jeffrey Chan, and Christopher Leckie. “Security Games for Virtual Machine Allocation in Cloud Computing”. In: *4th International Conference on Decision and Game Theory for Security*. 2013, pp. 99–118.
- [54] Khalid Bijon, Ram Krishnan, and Ravi Sandhu. “Mitigating Multi-Tenancy Risks in IaaS Cloud Through Constraints-Driven Virtual Resource Scheduling”. In: *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*. 2015, pp. 63–74.
- [55] Yi Han, Jeffrey Chan, Tansu Alpcan, and Christopher Leckie. “Using Virtual Machine Allocation Policies to Defend against Co-resident Attacks in Cloud Computing”. In: *IEEE Transactions on Dependable and Secure Computing* 14.1 (2015).

- [56] Huang Xiaowei. “Forensic Analysis in Access Control: a Case-Study of a Cloud Application.” MA thesis. University of Waterloo, 2019.
- [57] Rodney McKemmish. *What is forensic computing?* Australian Institute of Criminology Canberra, 1999.
- [58] Brian Carrier. *File system forensic analysis*. Addison-Wesley Professional, 2005.
- [59] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. “ReVirt: Enabling intrusion analysis through virtual-machine logging and replay”. In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 211–224.
- [60] Sriranjani Sitaraman and S. Venkatesan. “Forensic analysis of file system intrusions using improved backtracking”. In: *3rd IEEE International Workshop on Information Assurance*. Mar. 2005, pp. 154–163.
- [61] Ashvin Goel, Wu-chang Feng, David Maier, Wu-chi Feng, and Jonathan Walpole. “Forensix: a robust, high-performance reconstruction system”. In: *25th IEEE International Conference on Distributed Computing Systems Workshops*. June 2005, pp. 155–162.
- [62] Ningning Zhu and Tzi-Cker Chiueh. “Design, implementation, and evaluation of repairable file service”. In: *International Conference on Dependable Systems and Networks*. 2003, pp. 217–226.
- [63] Samuel T. King and Peter M. Chen. “Backtracking Intrusions”. In: *ACM Symposium on Operating Systems Principles*. 2003, pp. 223–236.
- [64] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 2nd. 2008.
- [65] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. “Protection in Operating Systems”. In: *Communications of the ACM* 19.8 (Aug. 1976), pp. 461–471.
- [66] Michael A. Harrison and Walter L. Ruzzo. “Monotonic protection systems”. In: *Foundations of Secure Computation* (1978), pp. 337–365.
- [67] Ninghui Li and William H Winsborough. “Beyond proof-of-compliance: Safety and availability analysis in trust management”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2003, pp. 123–139.
- [68] Ninghui Li, John C. Mitchell, and William H. Winsborough. “Beyond proof-of-compliance: security analysis in trust management”. In: *Journal of the ACM* 52.3 (2005), pp. 474–514.
- [69] Ninghui Li and Mahesh V. Tripunitara. “Security Analysis in Role-based Access Control”. In: *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies*. 2004, pp. 126–135.

- [70] Ninghui Li and Mahesh V Tripunitara. “On safety in discretionary access control”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2005, pp. 96–109.
- [71] Ninghui Li, John C. Mitchell, and William H. Winsborough. “Beyond Proof-of-compliance: Security Analysis in Trust Management”. In: *Journal of the ACM* 52.3 (May 2005), pp. 474–514.
- [72] Mahesh V. Tripunitara and Ninghui Li. “The Foundational Work of Harrison-Ruzzo-Ullman Revisited”. In: *IEEE Transactions on Dependable and Secure Computing* 10.1 (Jan. 2013), pp. 28–39.
- [73] Jon A. Solworth and Robert H. Sloan. “A layered design of discretionary access controls with decidable safety properties”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2004, pp. 56–67.
- [74] Paul E. Ammann and Ravi S. Sandhu. “Safety analysis for the extended schematic protection model”. In: *IEEE Computer Society Symposium on Research in Security and Privacy*. 1991, pp. 87–97.
- [75] Ravi S. Sandhu. “The schematic protection model: its definition and analysis for acyclic attenuating schemes”. In: *Journal of the ACM* 35.2 (1988), pp. 404–432.
- [76] Ravi S. Sandhu. “The typed access matrix model.” In: *Proceedings of the IEEE Symposium on Security and Privacy*. 1992, pp. 122–136.
- [77] Matt Blaze, Joan Feigenbaum, and Jack Lacy. “Decentralized trust management”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 1996, pp. 164–173.
- [78] Dorothy E. Denning. “An intrusion-detection model”. In: *IEEE Transactions on Software Engineering* 2 (1987), pp. 222–232.
- [79] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. “The ARBAC97 Model for Role-based Administration of Roles”. In: *ACM Transactions on Information and System Security* 2.1 (Feb. 1999), pp. 105–135.
- [80] G. Scott Graham and Peter J. Denning. “Protection: Principles and Practice”. In: *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*. 1972, pp. 417–429.
- [81] Someh Jha, Ningui Li, Mahesh Tripunitara, Qihua Wang, and William Winsborough. “Towards Formal Verification of Role-Based Access Control Policies”. In: *IEEE Transactions on Dependable and Secure Computing* 5.4 (Oct. 2008), pp. 242–255.
- [82] Nordstrom Technology. *Hello, Retail!* <https://github.com/Nordstrom/hello-retail>. Last accessed: April 2, 2020. Sept. 2018.



- [83] Amazon Web Services (AWS). *Serverless – Build and run applications without thinking about servers*. Accessed: 2020-04-02.
- [84] aws.amazon.com. *Amazon Web Services*. Accessed: 2020-04-02.
- [85] John McKim. *Announcing the Winners of the Inaugural Serverless Architecture Competition*. <https://read.acloud.guru/announcing-the-winners-of-the-inaugural-serverlessconf-architecture-competition-1dce2db6da3>. Last accessed: April 2, 2020.
- [86] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. 2nd. 2004.
- [87] Jonathan Shahren, Jianwei Niu, and Mahesh V. Tripunitara. “Cree: a Performant Tool for Safety Analysis of Administrative Temporal Role-Based Access Control (ATRBAC) Policies”. In: *IEEE Transactions on Dependable and Secure Computing* (2019).
- [88] Jonathan Shahren. “Automated Safety Analysis of Administrative Temporal Role-Based Access Control (ATRBAC) Policies using Mohawk+T”. MA thesis. University of Waterloo, 2016.
- [89] Silvio Ranise, Anh Truong, and Alessandro Armando. “Scalable and Precise Automated Analysis of Administrative Temporal Role-based Access Control”. In: *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*. 2014, pp. 103–114.
- [90] Emre Uzun, Vijayalakshmi Atluri, Shamik Sural, Jaideep Vaidya, Gennaro Parlato, Anna Lisa Ferrara, and Madhusudan Parthasarathy. “Analyzing Temporal Role Based Access Control Models”. In: *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*. 2012, pp. 177–186.
- [91] Mikhail I. Gofman, Ruiqi Luo, Ayla C. Solomon, Yingbin Zhang, Ping Yang, and Scott D. Stoller. “RBAC-PAT: A Policy Analysis Tool for Role Based Access Control”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Stefan Kowalewski and Anna Philippou. 2009, pp. 46–49.
- [92] Brian Carrier. “Defining Digital Forensic Examination and Analysis Tools Using Abstraction Layers”. In: *International Journal of Digital Evidence* (2003).
- [93] Sean Philip Peisert. *A model of forensic analysis using goal-oriented logging*. 2007.
- [94] Matt Bishop, Christopher Wee, and Jeremy Frank. *Goal-oriented auditing and logging*. 1996.
- [95] Amazon Web Services (AWS). *AWS CloudTrail – User Guide*. <https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-user-guide.html>. Accessed: 2020-04-14.

- [96] Amazon Web Services (AWS). *Amazon CloudWatch – Observability of your AWS resources and applications on AWS and on-premises*. <https://aws.amazon.com/cloudwatch/>. Accessed: 2020-04-14.
- [97] Amazon Web Services (AWS). *AWS Lambda – Run code without thinking about servers. Pay only for the compute time you consume*. <https://aws.amazon.com/lambda/>. Accessed: 2020-04-14.
- [98] Amazon Web Services (AWS). *AWS Identity and Access Management (IAM) – Securely manage access to AWS services and resources*. <https://aws.amazon.com/iam/>. Accessed: 2020-04-15.
- [99] Amazon Web Services (AWS). *Amazon S3 – Object storage built to store and retrieve any amount of data from anywhere*. <https://aws.amazon.com/s3/>. Accessed: 2020-04-14.
- [100] Amazon Web Services (AWS). *Amazon EC2 – Secure and resizable compute capacity in the cloud*. <https://aws.amazon.com/ec2/>. Accessed: 2020-04-15.
- [101] Charlie Osborne. *The top 10 security challenges of serverless architectures*. <https://www.zdnet.com/article/the-top-10-risks-for-apps-on-serverless-architectures/>. Accessed: 2020-04-15.
- [102] CloudSploit. *A Technical Analysis of the Capital One Hack*. <https://blog.cloudsploit.com/a-technical-analysis-of-the-capital-one-hack-a9b43d7c8aea>. Accessed: 2020-04-21.
- [103] Michael G. Noblett, Mark M. Pollitt, and Lawrence A. Presley. “Recovering and Examining Computer Forensic Evidence”. In: *Forensic Science Communications* 2.4 (Oct. 2000).
- [104] Sean Peisert, Sidney Karin, Matt Bishop, and Keith Marzullo. “Principles-driven forensic analysis”. In: *Proceedings of the 2005 workshop on New security paradigms*. 2005, pp. 85–93.
- [105] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. “Toward Models for Forensic Analysis”. In: *2nd International Workshop on Systematic Approaches to Digital Forensic Engineering*. 2007, pp. 3–15.
- [106] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. “Analysis of Computer Intrusions Using Sequences of Function Calls”. In: *IEEE Transactions on Dependable and Secure Computing* 4.2 (Apr. 2007), pp. 137–150.
- [107] Alan M. Christie. *The Incident Detection, Analysis, and Response (IDAR) Project*. Tech. rep. CERT Coordination Center, 2002.

- [108] M Tyson, P Berry, N Williams, D Moran, and D Blei. *DERBI: Diagnosis, explanation and recovery from computer break-ins*. Tech. rep. DARPA Project F30602-96-C-0295 Final Report, SRI International, Artificial Intelligence Center, 2001.
- [109] Martin S. Olivier. “On metadata context in database forensics”. In: *Digital Investigation* 5.3 (2009), pp. 115–123.
- [110] Maria Stoyanova, Yannis Nikoloudakis, Spyridon Panagiotakis, Evangelos Pallis, and Evangelos K Markakis. “A Survey on the Internet of Things (IoT) Forensics: Challenges, Approaches and Open Issues”. In: *IEEE Communications Surveys and Tutorials* (2020).
- [111] Bharat Manral, Gaurav Somani, Kim-Kwang Raymond Choo, Mauro Conti, and Manoj Singh Gaur. “A Systematic Survey on Cloud Forensics Challenges, Solutions, and Future Directions”. In: *ACM Computing Surveys* (2019), pp. 1–38.
- [112] Butler W. Lampson. “Protection”. In: *ACM SIGOPS Operating Systems Review* 8.1 (Jan. 1974), pp. 18–24.
- [113] Jarrod A. Roy, Farinaz Koushanfar, and Igor L Markov. “EPIC: Ending piracy of integrated circuits”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. 2008, pp. 1069–1074.
- [114] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z Pan, and Yier Jin. “AppSAT: Approximately deobfuscating integrated circuits”. In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust*. 2017, pp. 95–100.
- [115] Pramod Subramanyan, Sayak Ray, and Sharad Malik. “Evaluating the security of logic encryption algorithms”. In: *2015 IEEE International Symposium on Hardware Oriented Security and Trust*. 2015, pp. 137–143.
- [116] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2014. ISBN: 9781466570269.
- [117] Federal Information Processing Standards (FIPS). *Secure Hash Standard*. Publication 180-2. Aug. 2002.
- [118] Mohamed El Massad, Nahid Juma, Jonathan Shahren, Mariana Raykova, Siddharth Garg, and Mahesh Tripunitara. *Xor-Locked One-Way Functions and Strong Locked Circuits*. <https://ece.uwaterloo.ca/~tripunit/logiclockingacmtops2020/>. June 2020.
- [119] Muhammad Yasin, Abhrajit Sengupta, Benjamin Carrion Schafer, Yiorgos Makris, Ozgur Sinanoglu, and Jeyavijayan Rajendran. “What to lock? Functional and parametric locking”. In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. 2017, pp. 351–356.

- [120] Deepak Sirone and Pramod Subramanayan. “Functional analysis attacks on logic locking”. In: *IEEE Transactions on Information Forensics and Security* 15 (2020).
- [121] Fangfei Yang, Ming Tang, and Ozgur Sinanoglu. “Stripped Functionality Logic Locking With Hamming Distance-Based Restore Unit (SFLL-hd)–Unlocked”. In: *IEEE Transactions on Information Forensics and Security* 14.10 (2019), pp. 2778–2786.
- [122] Kaveh Shamsi, David Z. Pan, and Yier Jin. “On the impossibility of approximation-resilient circuit locking”. In: *2019 IEEE International Symposium on Hardware Oriented Security and Trust*. 2019, pp. 161–170.
- [123] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan Rajendran, and Ozgur Sinanoglu. “Provably-secure logic locking: From theory to practice”. In: *Proceedings of the 2017 ACM Conference on Computer and Communications Security*. 2017, pp. 1601–1618.
- [124] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. “On improving the security of logic locking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.9 (2015), pp. 1411–1424.
- [125] Hai Zhou. “A Humble Theory and Application for Logic Encryption.” In: *IACR Cryptology ePrint Archive* (2017), p. 696.
- [126] The Information Warfare Site. *Descriptions of SHA-256, SHA-384, and SHA-512*. <http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf>. Accessed: 2019-08-01.
- [127] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 2011. ISBN: 0716710455.
- [128] Stephen A. Cook and David G. Mitchell. “Finding Hard Instances of the Satisfiability Problem: A Survey”. In: 1997, pp. 1–17.
- [129] *Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification*. <http://www.eecs.berkeley.edu/~alanmi/abc/>.
- [130] Mihir Bellare and Igors Stepanovs. “Point-function obfuscation: a framework and generic constructions”. In: *Theory of Cryptography Conference*. 2016, pp. 565–594.
- [131] Kerckhoffs Auguste. “La Cryptographie Militaire”. In: *Journal des Sciences Militaires* (1883), p. 5.
- [132] Phillip Rogaway. “On the role definitions in and beyond cryptography”. In: *Annual Asian Computing Science Conference*. 2004, pp. 13–32.

- [133] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. "Security analysis of logic obfuscation". In: *Proceedings of the 49th Annual Design Automation Conference*. 2012, pp. 83–89.
- [134] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. "Security Analysis of Anti-Sat". In: *2017 22nd Asia and South Pacific Design Automation Conference*. 2017, pp. 342–347.
- [135] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. "Obfuscation of probabilistic circuits and applications". In: *Theory of Cryptography Conference*. 2015, pp. 468–497.
- [136] Dan Boneh and Brent Waters. "Constrained pseudorandom functions and their applications". In: *International Conference on the Theory and Application of Cryptology and Information Security*. 2013, pp. 280–300.
- [137] Mohamed El Massad, Siddharth Garg, and Mahesh V. Tripunitara. "Integrated Circuit (IC) Decamouflaging: Reverse Engineering Camouflaged ICs within Minutes." In: *Network and Distributed System Security Symposium (NDSS)*. 2015, pp. 1–14.
- [138] Alex Baumgarten, Akhilesh Tyagi, and Joseph Zambreno. "Preventing IC piracy using reconfigurable logic barriers". In: *IEEE Design and Test of Computers* 27.1 (2010), pp. 66–75.
- [139] Sophie Dupuis, Papa-Sidi Ba, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. "A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans". In: *2014 IEEE 20th International On-Line Testing Symposium*. 2014, pp. 49–54.
- [140] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. "Fault analysis-based logic encryption". In: *IEEE Transactions on Computers* 64.2 (2013), pp. 410–424.
- [141] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z Pan, and Yier Jin. "Cyclic obfuscation for creating sat-unresolvable circuits". In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. 2017, pp. 173–178.
- [142] Yang Xie and Ankur Srivastava. "Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.
- [143] Shervin Roshanisehat, Hadi Mardani Kamali, and Avesta Sasan. "SRCLock: SAT-resistant cyclic logic locking for protecting the hardware". In: *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. 2018, pp. 153–158.

- [144] Hai Zhou, Ruifeng Jiang, and Shuyu Kong. “CycSAT: SAT-based attack on cyclic logic encryptions”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design*. 2017, pp. 49–56.
- [145] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. “SMT attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the SAT attacks”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 97–122.
- [146] Kaveh Shamsi, Meng Li, David Z Pan, and Yier Jin. “Cross-lock: Dense layout-level interconnect locking using cross-bar architectures”. In: *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. 2018, pp. 147–152.
- [147] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. “Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks”. In: *Proceedings of the 56th Annual Design Automation Conference*. 2019, pp. 1–6.
- [148] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan JV Rajendran, and Ozgur Sinanoglu. “SARLock: SAT attack resistant logic locking”. In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust*. 2016, pp. 236–241.
- [149] Yang Xie and Ankur Srivastava. “Mitigating sat attack on logic locking”. In: *International conference on cryptographic hardware and embedded systems*. 2016, pp. 127–146.
- [150] Meng Li, Kaveh Shamsi, Travis Meade, Zheng Zhao, Bei Yu, Yier Jin, and David Z. Pan. “Provably secure camouflaging strategy for IC protection”. In: *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* (2017).
- [151] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. “Removal attacks on logic locking and camouflaging techniques”. In: *IEEE Transactions on Emerging Topics in Computing* (2017).
- [152] Yuanqi Shen and Hai Zhou. “Double dip: Re-evaluating security of logic encryption algorithms”. In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. 2017, pp. 179–184.
- [153] Xiaolin Xu, Bicky Shakya, Mark M Tehranipoor, and Domenic Forte. “Novel bypass attack and BDD-based tradeoff analysis against all known logic locking attacks”. In: *International Conference on Cryptographic Hardware and Embedded Systems*. 2017, pp. 189–210.
- [154] Dominik Šišejković, Rainer Leupers, Gerd Ascheid, and Simon Metzner. “A unifying logic encryption security metric”. In: *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. 2018, pp. 179–186.

- [155] Messaoud Benantar. *Access control systems: security, identity management and trust models*. 2005.

## A Forensic Analysis for Access Control

### A An Upper-bound on Hardness for General ARBAC

In Section 4.3.2 we presented a restricted version of the ARBAC scheme that sufficed for our lower-bound result in Theorem 4.3.3. Here, we describe a general version of the ARBAC scheme and show that the upper-bound result in Theorem 4.3.5 for the restricted version applies for this general version also.

**The Scheme** We adopt the ARBAC97 scheme [79] and give an exposition similar to prior work on safety analysis [81]. A system is associated with finite sets of users ( $U$ ), roles ( $R$ ) and permissions ( $P$ ). A state  $\gamma$  is a tuple  $\langle UA_\gamma, PA, RH \rangle$ , where  $UA_\gamma \subseteq U \times R$  is the user-role assignment relation,  $PA \subseteq P \times R$  is the permission-role assignment relation and  $RH$  is the role-hierarchy. The latter is specified by relations of the form  $r_1 \succeq r_2$  which denotes that the role  $r_1$  inherits all the permissions associated with the role  $r_2$ , and thus, members of  $r_1$  are implicitly also members of  $r_2$ .

Given a state,  $\gamma$ , each user has a set of roles for which the user is authorized. We formalize this by defining a function  $\text{authorizedRoles}(u) \rightarrow 2^R$ , that when applied to state  $\gamma$  evaluates as:

$$\text{authorizedRoles}(u) = \{r \in R \mid \exists r_1 \in R \text{ such that } [(u, r_1) \in UA_\gamma \wedge (r_1 \succeq r)]\}$$

We also define  $\text{down}(r)$  to be the set of all roles dominated by  $r$  and  $\text{up}(r)$  to be the set of all roles that dominate  $r$  as follows:

$$\text{down}(r) = \{r' \in R \mid r \succeq r'\}$$

$$\text{up}(r) = \{r' \in R \mid r' \succeq r\}$$

ARBAC defines administrative roles and specifies how members of these roles can change an RBAC policy. A policy can be changed by making modifications to the: (1) user-role assignment, (2) permission-role assignment, and (3) role hierarchy. As in prior work [81]



we focus on changes made to the user-role assignment, because it is the most volatile [155]. Changes allowed to the user-role assignment are described by three components  $CA$ ,  $CR$  and  $CO$ .

$CA \subseteq R \times C \times 2^R$  determines who can assign users to roles and the preconditions these users must satisfy.  $C$  is the set of conditions, which are expressions formed using roles, the binary operators  $\wedge$  and  $\vee$ , the unary operator  $\neg$ , and parentheses. A tuple  $\langle r_a, c, rset \rangle$  in  $CA$  means that members of the administrative role  $r_a \in R$  can assign any user whose role memberships satisfy the condition  $c$ , to any role  $r \in rset$ , where  $rset \subseteq R$ . For example,  $\langle r_0, r_1 \wedge r_2 \wedge \neg r_3, \{r_4\} \rangle \in CA$  means that a user that is a member of the role  $r_0$  is allowed to assign a user that is a member of both  $r_1$  and  $r_2$ , but is not a member of  $r_3$ , to be a member of  $r_4$ .

$CR \subseteq R \times 2^R$  determines who can remove users from roles. A tuple  $\langle r_a, rset \rangle \in CR$  means that the members of the administrative role  $r_a \in R$  can remove a user from a role  $r \in rset$ , where  $rset \subseteq R$ . Unlike the relation  $CA$ , there are no preconditions in the relation  $CR$ . We assume that  $CA$  and  $CR$  satisfy the property that the administrative roles are not affected by  $CA$  and  $CR$ . The administrative roles appear in the first component of each tuple in  $CA$  or  $CR$ . These roles do not appear in the last component of any  $CA$  or  $CR$  tuple. This condition is satisfied in ARBAC97, which assumes the existence of a set of administrative roles that is disjoint from the set of normal roles.

$CO$  is a set of mutually exclusive role constraints. Each constraint has the form  $\langle \{r_1, \dots, r_m\}, t \rangle$  where each  $r_i$  is a role in  $R$ , and  $m$  and  $t$  are integers such that  $1 < t \leq m$ . This constraint forbids a user from being a member of  $t$  or more roles in  $\{r_1, \dots, r_m\}$ . A set of roles  $R_i \subseteq R$  satisfies a constraint  $\langle \{r_1, \dots, r_m\}, t \rangle$  if and only if  $|R_i \cap \{r_1, \dots, r_m\}| < t$ . For example,  $\langle \{r_1, r_2\}, 2 \rangle$  means that no user is allowed to be a member of both  $r_1$  and  $r_2$ . For example, if in state  $\gamma$ ,  $r_1 \in \text{authorizedRoles}(u)$  for user  $u$ , then an assignment action that assigns  $u$  to any role in  $up(r_2)$  would fail because of the constraint.

$\Psi$  consists of a single state-change rule,  $\psi$ , which is a set of actions:

$$\psi = \{assign(u_a, u_t, r_t) \mid u_a, u_t \in U \wedge r_t \in R\} \cup \{revoke(u_a, u_t, r_t) \mid u_a, u_t \in U \wedge r_t \in R\}$$

An assignment action  $assign(u_a, u_t, r_t)$  means that the user  $u_a$  assigns the user  $u_t$  to the role  $r_t$ . When this action is applied to an RBAC state,  $\gamma$ , it succeeds if and only if the following three conditions hold:

1.  $(u_t, r_t) \notin UA_\gamma$ , i.e., user  $u_t$  is not already assigned to role  $r_t$ .
2. There exists a tuple  $\langle r_a, c, rset \rangle \in CA$  such that  $r_a \in authorizedRoles(u_a)$ ,  $authorizedRoles(u_t)$  satisfies  $c$ , and  $r_t \in rset$ .
3.  $authorizedRoles(u_t) \cup down(r_t)$  satisfies every constraint in  $CO$ , i.e., the new role memberships of  $u_t$  do not violate any constraint.

When the assignment action is successfully applied to an RBAC state,  $\gamma$ , the resulting state,  $\gamma'$ , differs from  $\gamma$  only in the user-role relation. The result of a successful application is  $UA_{\gamma'} = UA_\gamma \cup \{(u_t, r_t)\}$ . When the application is not successful, the state does not change.

A revocation action  $revoke(u_a, u_t, r_t)$  means that the user  $u_a$  revokes the user  $u_t$  from the role  $r_t$ . When this action is applied to an RBAC state,  $\gamma$ , it succeeds if and only if the following two conditions hold:

1.  $(u_t, r_t) \in UA_\gamma$ , i.e., user  $u_t$  is assigned to role  $r_t$ .
2. There exists a tuple  $\langle r_a, rset \rangle \in CA$  such that  $r_a \in authorizedRoles(u_a)$  and  $r_t \in rset$ .

When the revocation action is successfully applied to an RBAC state,  $\gamma$ , the resulting state,  $\gamma'$ , differs from  $\gamma$  only in the user-role relation. The result of a successful application is  $UA_{\gamma'} = UA_\gamma \setminus \{(u_t, r_t)\}$ . When the application is not successful, the state does not change.

## Upper-bound

**Theorem A.1.** *Forensic analysis for the general ARBAC scheme described above with  $\pi = possible$  and  $L = \emptyset$  is in **PSPACE**.*

*Proof.* Our proof is by construction. We propose a non-deterministic algorithm that is correct and is guaranteed to halt, and that consumes space at worst polynomial in the size of the input, thereby proving that the problem is in **NPSPACE**, the class of decision problems for which there exists a non-deterministic algorithm that consumes space only polynomial in the size of the input. We then appeal to Savitch's theorem, **NPSPACE** = **PSPACE** [7]. Given a forensics instance,  $\langle \gamma, \psi, k, q, \pi, L \rangle$ , where  $q = (u, r)$ , our algorithm “inverts” state-changes, and goes “backwards” starting at the current state,  $\gamma$ .

We observe that to decide whether a query  $(u, r)$ , may have been true in some state between  $k$  and  $\gamma$ , we do not need to consider changes to role-memberships of users other than  $u$ . This is because membership to administrative roles are not changed by  $\psi$ , and membership to non-administrative roles of users other than  $u$  do not effect  $u$ 's role-memberships.

We maintain a “present state,” call it  $p$ . Consider  $p = \langle UA_p, PA, RH \rangle$ . The  $PA$  and  $RH$  components of  $p$  are the same as those of  $\gamma$  and  $k$  of the input. As for the  $UA_p$  component, because we need to consider changes to  $u$ 's role-membership only, it suffices to record  $u$ 's present role-memberships. There are  $2^{|R|}$  possible role-membership configurations for  $u$ , thus the size of  $p$  is  $|R|$  bits.

We then enumerate all possible state-changes that could have resulted from an immediately prior state,  $p'$ , to  $p$ . This is our “inversion” of state-changes. There can be at most linearly, and therefore, polynomially, many such state-changes because the number is upper-bounded by the cardinality of  $\psi$ . To enumerate the possible state-changes, we do the following. For each assign action  $(u_a, u, r_t) \in \psi$ , we ask whether  $(u, r_t) \in UA_p$ , i.e.,  $u$  is a member of  $r_t$  in the state  $p$ . If yes, we go through the set  $CA$  to check if there is a tuple  $(r_a, c, rset)$  such that  $r_a \in \text{authorizedRoles}_p(u_a)$ ,  $\text{authorizedRoles}_p(u)$  satisfies  $c$ , and  $r_t \in rset$ . If yes, then that assign action was a possible state-change, with the prior state  $p'$  being identical to  $p$  except for  $UA'_p = UA_p \setminus (u, r_t)$ . Otherwise, it is not a possible state-change. Similarly, for each revoke action  $\langle u_a, u_t, r_t \rangle$ , we ask whether  $r_t \notin UA_p$ . If yes, we check whether  $\text{authorizedRoles}_p(u) \cap \text{down}(r_t)$  satisfies every constraint in  $CO$ . If yes, we go through the set  $CR$  to check if there is a tuple  $(r_a, rset)$  such that  $r_a \in \text{authorizedRoles}_p(u_a)$ , and  $r_t \in rset$ . If yes, then that revoke action was a possible state-change, with the prior state  $p'$

being identical to  $p$  except that  $UA'_p = UA_p \cup (u, r_t)$ .

Then from amongst the possible state-changes, we non-deterministically pick one, and set  $p$  to the corresponding prior state,  $p'$ . Apart from space for  $p$  and the enumeration of possible state-changes, we allocate space also for: (i) the number of “backwards” state-changes we have effected, and, (ii) a boolean for whether we have encountered a state  $p$  such that  $p \vdash q$ . The size of (i) is  $|R|$  bits. This is because the number of different states is upper-bounded by  $2^{|R|}$ , and therefore a counter of  $|R|$  bits suffices. The size of (ii) is one bit.

Every time we effect a (non-deterministic) state-change backwards with a new present state  $p$ , we do the following. We increment our counter (i) above. Then we check whether the bit (ii) above is already set. If yes, we check if  $authorizedRoles_p(u) = authorizedRoles_k(u)$ . If yes, we halt and output ‘true.’ If the bit (ii) is not set, we check whether the new state  $p \vdash q$ , i.e., whether  $r \in authorizedRoles_p(u)$ . If yes, we set the bit (ii) and we clear the state counter to 0. If we do not return ‘true’ after this state-change, then we check whether our counter has reached  $2^{|R|}$ . If yes, we immediately halt and output ‘false.’ Otherwise, we continue, i.e., enumerate possible state-changes with the new present state  $p$ .

The algorithm above is correct because there exists a sequence of non-deterministic choices that causes us to output ‘true’ if and only if the input instance is indeed true. Also, the algorithm is guaranteed to halt because of the counter we maintain. The total space we need is at worst linear, and therefore polynomial, in the size of the input.  $\square$

## B Graham-Denning with Trusted Users

We present in Algorithm 6 a polynomial-time procedure to solve forensic analysis for Graham-Denning based systems when  $\pi = \text{possible}$ ,  $L = \emptyset$ , and trusted users are considered. Algorithm 6 makes the following assumptions: (1) The universal subject  $u$  is trusted, (2) the past known state  $k$  contains at least one untrusted subject, (3) the subject  $s$  and object  $o$  in the input query  $(s, o, x)$  exist in the current state  $\gamma$ . Observe that if (2) does not hold, then the instance cannot be true.

**Input:** Forensic instance,  $\langle i, \gamma, \psi, q = (s, o, x), \text{possible}, T \rangle$

**Output:** true or false

```

1 if  $x = \text{control}$  then output false
2 if  $x = \text{own} \wedge o \in S_\gamma$  then
3   if  $o \notin S_k$  then output true
4   if  $o \in S_k \wedge \text{SubjectOwnerChain}(o, k, \gamma, T) = \text{true} \wedge \text{BreakCyle}(s, o, k, T) = \text{true}$ 
      then output true
5 if  $x = \text{own} \wedge o \in O_\gamma \setminus S_\gamma$  then output false
6 if  $x \in R \cup R^*$  then
7    $y = x$ 
8   if  $x \in R$  then  $y = x^*$ 
9   if  $\exists s_u \in S_\gamma \setminus T$  s.t.  $y \in M_\gamma[s_u, o] \vee \text{own} \in M_\gamma[s_u, o]$  then output true
10  if  $o \in O_k \wedge \exists s_u \in S_k \setminus T$  s.t.  $y \in M_k[s_u, o] \vee \text{own} \in M_k[s_u, o]$  then output true
11  if  $o \in S_\gamma$  then
12    if  $o \notin S_k$  then output true
13    if  $o \in S_k \wedge \text{SubjectOwnerChain}(o, k, \gamma, T) = \text{true}$  then output true
14  if  $o \in O_\gamma \setminus S_\gamma$  then
15    if  $o \notin S_k \wedge \exists s_u \in S_k \setminus T$  s.t.
       $\text{UntrustedSubjectOwnerChain}(s_u, k', \gamma, T) = \text{true}$  then output true
16    if  $o \in S_k \wedge \text{ObjectOwnerChain}(o, k, \gamma, T) = \text{true}$  then output true
17 output false

```

**Algorithm 6:** Algorithm to solve forensic analysis for Graham-Denning, when  $\pi = \text{possible}$ ,  $L = \emptyset$ ,  $q = (s, o, x)$ ,  $o \in O_\gamma$ ,  $s \in S_\gamma$ ,  $k \not\models q$  and  $\gamma \not\models q$ . In line 15,  $k'$  is identical to  $k$  except it has a column for  $o$  and  $\text{own} \in M'_k[s_u, o]$

Below are terms and functions used in Algorithm 6:

- A *non-subject object* is an object that is not a subject.
- An *exclusive ownership chain* of an object  $o$  in state  $\gamma$  is an ordered set  $\{o_1, o_2, \dots, u\}_\gamma$  where  $o_1$  owns  $o$ ,  $o_2$  owns  $o_1$  etc., in  $\gamma$ . Note that  $o$  is not included in  $o$ 's exclusive ownership chain.
- An *inclusive ownership chain* of an object  $o$  in state  $\gamma$  is an ordered set  $\{o, o_1, o_2, \dots, u\}_\gamma$  where  $o_1$  owns  $o$ ,  $o_2$  owns  $o_1$  etc., in  $\gamma$ . Note that the inclusive ownership chain of  $o$  includes  $o$  and it is the first object in the set.
- $OwnObjects(s, k, \gamma, C)$  takes as input a subject  $s$ , two states  $k$  and  $\gamma$ , and an ownership chain  $C$ . It returns true if and only if  $s \in C$ ,  $s \in S_\gamma$ , and in  $\gamma$ ,  $s$  has ownership over all objects in the set  $O'$  which is defined as follows.  $O'$  comprises of non-subject objects that exist in both states  $k$  and  $\gamma$ , and in the state  $k$ , each of these objects is owned by at least one subject from the subjects preceding  $s$  in the ownership chain  $C$ .
- $SubjectOwnerChain(s, k, \gamma, T)$  takes as input a subject  $s$ , states  $k$  and  $\gamma$ , and a set of trusted subjects  $T$ . It returns true if and only if  $s \in S_k$  and the exclusive ownership chain  $C$  of  $s$  in  $k$  is such that: (1) it contains an untrusted subject  $s^*$ , and (2) there exists a subject  $s'$  which is either  $s^*$  or one of the subjects that come after it in  $C$  such that  $s' \in S_\gamma$ , the subjects preceding  $s'$  in  $C$  do not exist in  $S_\gamma$ , and,  $OwnObjects(s', k, \gamma, C)$  is true.  $SubjectOwnerChain(s, k, \gamma, T)$  returning true on some input implies the possibility of  $s^*$  having destroyed the subjects preceding it in  $C$ , obtaining direct ownership over  $s$ , and then (later)  $s^*$  being destroyed by  $s'$  if  $s^* \neq s'$ .
- $UntrustedSubjectOwnerChain(s, k, \gamma, T)$  takes as input a subject  $s$ , states  $k$  and  $\gamma$ , and a set of trusted users  $T$ . It returns true if and only if  $s \notin T$ ,  $s \in S_k$  and the inclusive ownership chain  $C$  of  $s$  in  $k$  contains a subject  $s^*$  such that: (1)  $s^*$  exists in  $\gamma$ , (2) all subjects preceding  $s^*$  in  $C$  do not exist in state  $\gamma$ , and (3)  $OwnObjects(s^*, k, \gamma, C)$  is true. This function returning true on some input implies the possibility of either  $s$  being  $s^*$  or  $s^*$  having destroyed  $s$ .

- *ObjectOwnerChain*( $o, k, \gamma, T$ ) takes as input an object  $o$ , states  $k$  and  $\gamma$ , and a set of trusted subjects  $T$ . It returns true if and only if  $o \in O_k$  and from among all the exclusive ownership chains of  $o$  in  $k$ , at least one chain  $C$  is such that (1) it contains an untrusted subject  $s^*$ , and (2) there exists a subject  $s'$  which is either  $s^*$  or one of the subjects that come after it in  $C$  such that  $s'$  exists in  $S_\gamma$  and the subjects preceding  $s'$  in  $C$  do not exist in  $S_\gamma$ , and, *OwnObjects*( $s', k, \gamma, C$ ) is true. This function returning true on some input implies the possibility of  $s^*$  having destroyed the subjects preceding it in  $C$ , obtaining direct ownership over  $o$  and then (later),  $s^*$  being destroyed by  $s'$  if  $s^* \neq s'$ .
- *BreakCycle*( $s_1, s_2, k, T$ ) takes as input two subjects  $s_1$  and  $s_2$ , a state  $k$  and a set of trusted subjects  $T$ . It returns true if and only if one of two conditions hold: (1)  $s_2$  is not in  $s_1$ 's exclusive ownership chain in  $k$  or (2)  $s_2$  is in  $s_1$ 's exclusive ownership chain but one of the subjects preceding  $s_2$  in the chain is untrusted. This function returning true on some input implies that  $s_1$  can own  $s_2$  without violating property 7 of Graham-Denning states (see Section 4.3.3). To see why this is so when condition (2) holds, assume the untrusted subject in the ownership chain is  $s_u$  and the subject immediately preceding it in the chain is  $s_w$ .  $s_u$  can create a new subject  $s_v$  and (temporarily) transfer ownership over  $s_w$  to  $s_v$  to break the chain.

Below is a line-by-line explanation of Algorithm 6:

**[Line 1]** If  $x = \text{control}$ , the query is false because  $s \in S_\gamma$ ,  $o \in O_\gamma$ ,  $\gamma \not\vdash q$ , and the control access cannot be transferred or deleted (see Figure 4.5).

**[Lines 2-4]** If  $x = \text{own}$  and  $o$  is a subject, then there are 2 cases to consider: (A)  $o \notin S_k$ , (B)  $o \in S_k$ . In case (A) the query is true, because an untrusted user in  $S_k$  can create  $o$  and thereby own it, and then can transfer ownership to  $s$  (possibly creating  $s$  first if it doesn't already exist).  $s$  can then transfer ownership away. In case (B) the query is true if and only if 2 conditions hold in state  $k$ . The first is that *SubjectOwnerChain*( $o, k, \gamma, T$ ) is true and the second is that *BreakCycle*( $s, o, k, T$ ) is true. Because the untrusted subject in the ownership chain of  $o$  in  $k$  can gain direct ownership over  $o$  and then transfer it to  $s$ .

**[Line 5]** If  $x = \text{own}$  and  $o$  is a non-subject object the query is false because when a subject

owns a non-subject object, the ownership is retained until either the subject or the object is destroyed.

**[Lines 6-10]** If  $x \in R \cup R^*$ , then if  $x \in R$ , let  $y = x^*$ , else let  $y = x$ . That is,  $y \in R^*$ . An important observation is that if  $s$  obtains the right  $x$  over  $o$ , then because every subject controls itself,  $s$  can issue the command  $delete_x(s, s, o)$ . The query is true if any of the following two conditions hold because the untrusted subject  $s_u$  can grant or transfer  $x$  over  $o$  to  $s$ . (1) There exists an untrusted subject  $s_u$  in  $S_\gamma$  such that  $y \in M_\gamma[s_u, o]$  or  $own \in M_\gamma[s_u, o]$ . (2)  $o \in O_k$  and there exists an untrusted subject  $s_u$  in  $S_k$  such that  $y \in M_k[s_u, o]$  or  $own \in M_k[s_u, o]$ .

**[Lines 11-13]** If  $x \in R \cup R^*$ , and neither of the above 2 conditions hold, the algorithm evaluates based on whether  $o$  is a subject or not. If  $o$  is a subject, then there are two cases to consider: (A)  $o \notin S_k$ , (B)  $o \in S_k$ . In case (A) the query is true because an untrusted subject  $s_u$  in  $S_k$  can create the subject  $o$  and thereby own it.  $s_u$  can then grant  $y$  to  $s$  (possibly creating  $s$  first if it doesn't exist in  $k$ ). Finally,  $s_u$  can transfer ownership to  $o$ 's owner in  $\gamma$ . In case (B), the query is true if and only if  $SubjectOwnerChain(o, k, \gamma, T)$  is true because this implies there exists an untrusted subject in the ownership chain of  $o$  in  $k$  who can gain direct ownership over  $o$  and then grant  $x$  to  $s$  (possibly creating  $s$  first if it doesn't exist in  $k$ ). The untrusted subject can then transfer ownership away to the owner of  $o$  in  $\gamma$ .

**[Lines 14-16]** If  $o$  is a non-subject object, there are two cases to consider: (A)  $o \notin S_k$ , and (B)  $o \in S_k$ . In case (A), the query is true if and only if there exists an untrusted subject  $s_u \in S_k$  such that  $UntrustedSubjectOwnerChain(s_u, k', \gamma, T)$  is true, where  $k'$  is identical to  $k$  except there exists a column for object  $o$  and  $own \in M'_k[s_u, o]$ . This is because  $s_u$  can create  $o$ , and grant  $x$  over  $o$  to  $s$ . In case (B), the query is true if and only if  $ObjectOwnerChain(o, k, \gamma, T)$  is true.